Efficient Proof of RAM Programs from Any Public-Coin Zero-Knowledge System

Cyprien Delpech de Saint Guilhem   Emmanuela Orsini   Titouan Tanguy
Michiel Verbauwhede

imec-COSIC, KU Leuven, Belgium
firstname.lastname@kuleuven.be

Abstract

We present the first constant-round and concretely efficient zero-knowledge proof protocol for RAM programs using any *stateless* zero-knowledge proof system for Boolean or arithmetic circuits. Both the communication complexity and the prover and verifier run times asymptotically scale linearly in the size of the memory and the run time of the RAM program; we demonstrate concrete efficiency with performance results of our C++ implementation. At the core of this work is an arithmetic circuit which verifies the consistency of a list of memory access tuples in zero-knowledge. Using this circuit, we construct a protocol to realize an ideal array functionality using a generic stateless and public-coin zero-knowledge functionality. We prove its UC security and show how it can then be expanded to provide proofs of entire RAM programs. We also extend the Limbo zero-knowledge protocol (ACM CCS 2021) to UC-realize the zero-knowledge functionality that our protocol requires. The C++ implementation of our array access protocol extends that of Limbo and provides interactive proofs with 40 bits of statistical security with an amortized cost of 0.42ms of prover time and 2.8KB of communication per memory access, indenpendently of the size of the memory; with multi-threading, this cost is reduced to 0.12ms and 1.8KB respectively. This performance of our *public-coin* protocol approaches that of *private-coin* protocol BubbleRAM (ACM CCS 2020, 0.15ms and 1.5KB per access).

1 Introduction

A zero-knowledge (ZK) proof is a fundamental cryptographic tool which proves that a statement is true without revealing any other information. Since their introduction by Goldwasser, Micali and Rackoff [GMR85], ZK proofs have had a significant impact on cryptography and been the object of intense research work due to their theoretical importance and varied applicability.

Many types of ZK proof systems exist, each presenting different trade-offs between several efficiency measures. While in blockchain applications, the main focus is on succinct proofs of small statements [GGPR13, Gro16, XZZ+19, Set20], another line of research has focused on prover efficiency [JKO13, AHIV17, KKW18, BCR+19, DIO21, DOT21], while other works have successfully constructed ZK proof systems for very large statements with good concrete efficiency [WYKW21, YSWW21, WYX+21, BMRS21].

Unfortunately, these works focus mostly on statements represented as circuits, either Boolean or arithmetic, which can incur a significant overhead to prove properties of large statements that are more naturally represented as random-access machine (RAM) programs. Many interesting functions and applications, such as private database search or verification of program execution, greatly benefit from RAM-based expression where their running time can be sublinear in the data size.

Several recent works [HMR15, MRS17, BCG+18, HK20, FKL+21, HK21, HYD+21] have initiated the study of ZK proof systems for RAM programs, mainly focusing on the designated-verifier setting, also known as private-coin setting. Here, we instead turn to the following question:

> *Can we design a RAM-based ZK proof system with concrete efficiency in the public-coin setting?*

1.1 Contribution

To answer, this work presents the first *public-coin, constant-round and constant-overhead* , in both running time and communication complexity, ZK proof system for RAM programs *over any field*. Our starting point is the recent works by Franzese et al. [FKL+21] and Bootle et al. [BCG+18] which propose a different approach to RAM-based ZK protocols compared to previous works. In particular, they replace the need for a sorting network—used for example in the TinyRAM framework to avoid the use of oblivious RAM (ORAM) [BCG+13, BCTV14]—by a polynomial-based permutation check to ensure consistency of memory access.

Public-coin constant-overhead zero-knowledge in the RAM model of computation over any field. Our protocols take inspiration from the work of Franzese et al. which achieves concretely-efficient linear communication complexity and running time for both prover and verifier and that of Bootle et al. which achieves asymptotically superconstant prover computation and sublinear communication and verifier running time. The core of the construction is a protocol Π_{ZKArray} for private read and write access that uses a *stateful* circuit-based ZK functionality which can reactively re-use inputs for different proofs in the private-coin setting.

First, we modify Π_{ZKArray} to provide *stateless* ZK proofs. This allows for instantiations with prover-efficient public-coin systems, like those based on the MPC-in-the-Head framework [IKOS07]. Secondly, we generalize the protocol over any field, binary or prime. Note that both of these modifications lead to non-trivial changes in Π_{ZKArray} to achieve a final protocol with constant overhead.

More precisely, to realize a stateless ZK functionality we present a 'new circuit compilation' approach which, given a list of array accesses, creates a circuit C_{check} which verifies the list's consistency. The final C_{check} circuit is composed only of standard arithmetic gates and can be given as input to a generic circuit-based ZK functionality $\mathcal{F}_{\mathsf{ZK}}$. However, since the execution of C_{check} requires new inputs from the prover to perform the checks on the accesses, we adapt $\mathcal{F}_{\mathsf{ZK}}$ to accept circuits evaluated on inputs both fresh and stored previously.

When we generalise Π_{ZKArray} to also work with prime fields, the naïve approach of performing equality and comparison tests leads to a non-constant overhead. To avoid this issue, we describe a new ZK protocol for equality testing and over a prime field that could be of independent interest; we also describe a bound-checking protocol reminiscent of the range relation proof of [BCG+18]. These two protocols take advantage of both the new inputs given to C_{check} and of a permutation check similar to the one of Franzese et al. [FKL+21]—which dates back to [Nef01]. We also extend this permutation check to handle permutations of tuples, and not only of elements, by using an inner-product compression technique, similarly to Bootle et al.'s [BCG+18]. This is different to the packing technique used by Franzese et al. which works efficiently for binary fields but is too costly for prime fields of large characteristic.

Finally, we show how to extend our $\mathcal{F}_{\mathsf{ZKArray}}$ functionality to accept richer circuits and implement ZK protocols for RAM-based computation. We stress that our construction not only public-coin but also constant-round, unlike that of Bootle et al. [BCG+18], and can be made fully non-interactive using standard techniques.

Instantiation with MPC-in-the-Head protocols. We give a concrete instantiation of our general construction using the MPC-in-the-Head-based ZK protocol, Limbo described by Delpech de Saint Guilhem et al. [DOT21]. We chose this framework since, among other public-coin systems, it offers concrete overall efficiency, which makes such schemes competitive even for relatively large statements. Moreover, they offer linear prover and communication complexity, great flexibility in the choice of parameters and post-quantum security.

We stress that the choice of Limbo was due to its efficient prover running time, but other protocols such as KKW [KKW18] or Ligero [AHIV17] can also realize our required ZK functionality, with only minor modifications. Instead of favouring running time, we could improve the communication complexity of our construction by using Ligero instead which achieves sub-linear communication and shortest proof size, especially for very large circuits.

Implementation and efficiency results. Finally, we implement our protocols, and compare our results with related work. Our implementation shows that we can indeed achieve a RAM-based ZK system with both concrete and asymptotic efficiency in the public-coin setting. We observe that each RAM access we make is equivalent to proving 24 multiplication gates. In practice, when working over the prime field $GF(2^{61} - 1)$ we achieve an amortized cost of 0.2ms and 4.1KB for each RAM access. As far as we know this is the best result to date in the public-coin setting, and is comparable to the BubbleRAM protocol [HK20] which heavily relies on the private-coin nature of their underlying ZK protocol. However, more recent ZK proof for RAM programs, also in the setting of a designated verifier, have already superseded BubbleRAM; most notably its direct successor PrORAM [HK21] as well as the work by Franzese et al. [FKL+21] that greatly outperforms BubbleRAM in both communication and running time.

In the light of the rapid development of this line of work, we believe that our construction can be an important step forward in order to bridge the gap between private and public coin ZK protocols in the RAM model of computation. The details of our implementation and further comparison with other works can be found in Section 6.

1.2 Additional Related Work

We mainly compare our results with the work of Franzese et al. [FKL+21], which instantiate their construction with the VOLE-based ZK protocol QuickSilver [YSWW21], with the advantage of having a very efficient underlying ZK protocol in the private-coin setting which naturally supports conversion between Boolean and arithmetic authenticated values with no extra costs, and can rely on stateful zero-knowledge functionalities.

To the best of our knowledge, all known concretely efficient protocols on ZK for RAM programs are in the private-coin setting and use different techniques compared to our construction. In particular, the line of work started with BubbleRAM [HK20, HK21, HYD+21] relies on the use of garbled circuit ZK protocols, in the JKO-framework [JKO13], and achieves a non-constant overhead cost per memory access either due the use of ORAM or routing network. The work of Bootle et al. [BCG+18] does not appear to have been implemented, and despite is sublinear asymptotic performance, is not recommended for implementation by its authors due to large constants in the big-O notation. We therefore do not take it into account for our performance comparisons.

Concurrently to this work, the Cairo architecture was proposed as a practically efficient, Turing-complete and STARK-friendly architecture [GPR21]. The high-level approach taken in that work is similar to ours: the authors propose an architecture which can provide proofs of execution for any compatible program. However, their work is directed at proof systems based on sets of polynomial equation constraints and not at systems based on arithmetic circuits. Therefore their architecture is best applied with STARK-like proof systems which offer very different efficiency balances from our objective in this work.

1.3 Technical Overview

Our main contribution is an approach tailored to MPC-in-the-Head to check the consistency of a series of T read or write accesses to an initial array M of size N, using an arithmetic *checking circuit* C_{check} over an arbitrary field $\mathbb{F}$. By using specially designed sub-circuits for equality checks, bound checks and permutation checks, this circuit removes the need for any bit-decomposition, which is expensive in prime fields, to perform these operations. These sub-circuits are arranged to verify the consistency of a list $\mathcal{L}$ of access tuples which contains both the initial array, encoded as N tuples, and the accesses performed as T additional tuples.

We denote by $[x]$ wire values in C_{check} that are sensitive and should not be revealed when C_{check} is proven in zero-knowledge. The initial array M is encoded as a list $\mathcal{M} = ((i, i, \mathsf{write}, [M_i])_{i\in[N]}$. Each access then is encoded as a tuple of the form $([l], t, [\mathsf{op}], [d])$, where l denotes the index of the memory being accessed; t is a global timestamp unique to this access, initialized at N; $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$ denotes the type of access, which we identify $\mathsf{read} = 0 \in \mathbb{F}$ and $\mathsf{write} = 1 \in \mathbb{F}$; and d denotes the value being accessed. Given this, the circuit takes as initial input a list $\mathcal{L}$ which contains the N initial array values, encoded as $\mathcal{M}$, followed by the T access tuples (ordered according to $t \in [N+1, N+T]$). The circuit C_{check} verifies the consistency of the accesses by checking that every read access returns the last value written to the same address.

To do so, following the same approach by Franzese et al., it requires a second list $\mathcal{L}'$ that is a permutation of the initial list $\mathcal{L}$ with the difference that it is sorted first according to the address l, and then according to the timestamp t. That is, within $\mathcal{L}'$, all accesses to the same address are grouped together, and then sorted chronologically. Given such a list $\mathcal{L}'$, the circuit checks for the following criteria:

1. $\mathcal{L}'$ is a permutation of $\mathcal{L}$.

2. Every adjacent pair of access tuple concerns either the same address, or two adjacent ones; i.e. for $([l'_i], [t'_i], *, *)$ and $([l'_{i+1}], [t'_{i+1}], *, *)$ in $\mathcal{L}'$, it holds that

$$\big((l'_i = l'_{i+1}) \wedge (t'_i < t'_{i+1})\big) \vee (l'_i + 1 = l'_{i+1}).$$

3. All accesses are made to addresses within bounds; i.e. $l'_{N+T} = N$. (Combined with the adjacency requirement from the previous step this implies all addresses are bounded by N.)

4. All operations are either reads or writes; i.e. $\mathsf{op}_i \in \{0, 1\}$ for $i \in [N+T]$.

5. All read tuples contain the same value as the last one to be written at that address; this is checked pair-wise by evaluating

$$(l'_i + 1 = l'_{i+1}) \vee (d'_i = d'_{i+1}) \vee (\mathsf{op}_{i+1} = \mathsf{write}).$$

The differences with the check performed by the protocol of Franzese et al. are three fold. First, all of our checking circuit is arithmetic whereas only criteria 1, i.e. the permutation check, is performed with an arithmetic circuit in [FKL+21]. Second, we do not pack our values ahead of the permutation check as this would require operations over $\mathbb{F}^4$ which would be too big for fields of high prime characteristic; instead we use an inner-product compression technique to reduce this to the one-dimensional case. Finally, we do not check that the first access at every address is a write operation since this is enforced by the structure of $\mathcal{M}$ within $\mathcal{L}$; and we also additionally check that op_i is a bit which is not necessary in [FKL+21] as they work with fields of characteristic 2.

In order to evaluate these consistency criteria, we present three arithmetic sub-circuits, EqCheck, BdCheck and PermCheck, which respectively verify equality, upper and lower bounds, and permutation of sensitive values while preserving zero-knowledge. A detailed description of these circuits is given in Section 3. As outlined above, only the equivalent of PermCheck is computed as an arithmetic circuit by Franzese et al. These three circuits are designed using standard arithmetic operations (addition, multiplication and equality check against a public constant) and also contain the following additional commands.

- `Input`: this command allows the prover to give additional inputs to C_{check}, such as the permuted version of an array. We include it in the description of the sub-circuits to highlight that some additional inputs are required at certain points. As the prover is free to input arbitrary values, those inputs, which must satisfy certain properties, must then also be checked.

- `Rand`: this command produces one or more uniformly random elements of $\mathbb{F}$. It represents randomness needed for statistical verification of properties (such as polynomial equality). Looking ahead, such randomness must be produced only after the inputs of the circuits have been committed to so that they cannot be selected such that verification incorrectly succeeds with non-negligible probability.

Organization. After preliminaries on zero-knowledge and commitment functionalities, MPC-in-the-Head protocols and RAM-based computation in Section 2, Section 3 presents and analyzes our three sub-circuits and the final C_{check} circuit. Section 4 then presents our Π_{ZKArray} protocol and the functionalities that it uses and realizes; it also presents how these can be extended to realize ZK proofs of RAM programs. Section 5 presents a generalization of the Limbo protocol for the UC framework and shows that is realizes the $\mathcal{F}_{\mathsf{ZKin}}$ functionality required by the Π_{ZKArray} protocol. Finally, Section 6 discusses our C++ implementation and the results that we obtained.

2 Preliminaries

We use bold letters to denote vectors and matrices, e.g., $\mathbf{a}$, $\mathbf{B}$; the operator $*$ denotes the inner product of two vectors. We denote by $[d]$ the set of integers $\{1, \ldots, d\}$, and by $[e, d]$ the set of integers $\{e, \ldots, d\}$ with $1 < e < d$. The notation $\langle \cdot \rangle$ stands for secret-shared values, and $\langle \cdot \rangle_i$ is used for the share held by party P_i; the notation $[\,\cdot\,]$ denotes sensitive data that should not be publicly revealed.

2.1 Zero-Knowledge and Commitment Functionalities

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

The functionality runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$.
It is parametrized by $\mathsf{type} = \{\mathsf{Boolean}, \mathsf{Arithmetic}\}$

Proof: On input $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C, \mathsf{type}, x)$ from the prover, compute $y = C(x)$, send (Prove, C) to $\mathcal{V}$ and $\mathcal{S}$ and store (C, y).

Verify: On receiving $(\mathsf{sid}, \mathsf{Verify}, C, \mathsf{type})$ from $\mathcal{V}$, query $\mathcal{S}$; if $\mathcal{S}$ returns fail, or if (C, y) has not been stored, send $(\mathsf{sid}, C, 0)$ to $\mathcal{V}$. Otherwise, send (sid, C, y) to $\mathcal{V}$.

Figure 1: Ideal functionality for circuit-based ZK proofs

The protocols presented in this work are proven secure in the Universal Composability framework of Canetti [Can01]. We recall that given an NP relation $\mathcal{R}$, with corresponding language $\mathcal{L}$, for all valid instance $x \in \mathcal{L}$ there exists a string w (witness) such that $\mathcal{R}(x, w) = 1$; and for all invalid instance $x \notin \mathcal{R}$, then $\mathcal{R}(x, w) = 0$ for all strings w.

In a zero-knowledge proof, a prover $\mathcal{P}$ will prove to a verifier $\mathcal{V}$ that some NP statement x is true, using a valid witness w without leaking any additional information other than the veracity of the statement. A standard circuit-based zero-knowledge functionality is given in Figure 1, where C is an arithmetic or Boolean circuit such that $C_x(w) = 1 \iff \mathcal{R}(x, w) = 1$.

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

$\mathcal{F}_{\mathsf{Commit}}$ interacts with a sender $\mathcal{P}$, a receiver $\mathcal{V}$ and an adversary $\mathcal{S}$.
Let $\mathcal{M}$ be the space of valid messages.

Commit: Upon receiving $(\mathsf{sid}, \mathsf{Commit}, m)$ from $\mathcal{P}$, where $m \in \mathcal{M}$, store (sid, m) and send $(\mathsf{sid}, \mathsf{Committed}, [m])$ to $\mathcal{P}$, $\mathcal{V}$ and $\mathcal{S}$.

Open: Upon receiving $(\mathsf{sid}, \mathsf{Open}, [m])$ from $\mathcal{P}$ and $\mathcal{V}$, where (sid, m) was previously stored, send $(\mathsf{sid}, \mathsf{Open}, m)$ to $\mathcal{V}$.

Figure 2: Functionality $\mathcal{F}_{\mathsf{Commit}}$ for commitment

Figure 2 also presents the commitment functionality $\mathcal{F}_{\mathsf{Commit}}$ which returns handles to the committed values so that they can be selectively opened.

2.2 MPC-in-the-Head

In [IKOS07], Ishai, Kushilevitz, Ostrovsky and Sahai introduced the MPC-in-the-Head framework to build zero-knowledge proofs for NP-relations from secure multiparty computation. Let $\mathcal{P}$ be a prover and $\mathcal{V}$ a verifier with common input the statement x, and $\mathcal{P}$'s private input the witness w; and let f be the function which checks if w is a valid witness, i.e. $f_x(w) = \mathcal{R}(x, w)$. At a high level, an MPC-in-the-Head protocol will work as follows: the prover emulates "in its head" an MPC protocol between n parties that computes f, i.e. $\mathcal{P}$ generates a sharing $\langle w \rangle$ of the witness and distributes the corresponding shares as private inputs to the parties, and then simulates the evaluation of $f(\langle w \rangle) = \mathcal{R}(x, \langle w \rangle)$ by choosing uniformly random coins r_i for each party P_i, $i \in [n]$.

This emulation yields one transcript of the protocol execution per party. After this "MPC evaluation", $\mathcal{P}$ and $\mathcal{V}$ can interact to reveal a subset of transcripts, which the verifier can check for consistency. If the consistency check succeeds, then the verifier will be convinced that the prover knows w. Intuitively, the privacy of the MPC protocol ensures that this procedure does not leak any information about the witness if not too many transcripts are revealed.

Limbo. In our work we consider Limbo, an efficient instantiation of the MPC-in-the-Head framework which was described by Delpech de Saint Guilhem et al. [DOT21] and achieves good concrete prover performance, including for medium-large circuits.

Recall that Limbo is constructed from a client-server ρ-round MPC protocol Π_f, for a function f as described above, between a sender client P_S, computation servers $P_1, \ldots, P_n$, and a receiver client P_R. The authors present a zero-knowledge interactive oracle proof (ZK-IOP) protocol for arithmetic or Boolean circuit satisfiability based in part on a multiplication-checking MPC protocol, MultCheck, provided in [DOT21, Section 4.2].

The client-server MPC protocol used by the ZK-IOP protocol can be divided in the following two phases. First, the sender client P_S sends the inputs of the circuit to the servers, together with the outputs of every multiplication gate; using these, the servers perform a local computation of the circuit with secret-shared values. In the second phase, the servers use MultCheck to verify that P_S sent correct multiplication gate outputs. To do so, they first package the multiplication tuples[1] into randomised inner-product tuples using a public random-coin functionality. These inner-product tuples are then compressed repeatedly, again using a public random-coin functionality, until a single tuple of low dimension is left to be verified. This is done by P_R who receives every secret share of the tuple from the servers and can output 0 or 1 based on its correctness. To amplify the soundness this basic protocol needs to be repeated a certain number of times. In the paper, the authors show an improvement to this naïve approach.

Theorem 1 ([DOT21]) *If δ is the probability that* MultCheck *fails, i.e., an incorrect triple remains undetected, the basic version of* Limbo*, with τ repetitions, is a (honest-verifier) ZK-IOP with soundness error*

$$\epsilon = (1/n + (1 - 1/n) \cdot \delta)^{\tau}.$$

2.3 RAM-based Computation

We follow the RAM-based computation model described by Gordon et al. [GKK+12]. We focus on RAM program for computing a function $f(x, M)$, where x is a small, possibly public, input, and M is a large dataset, which can be viewed as stored in a memory array $M_1, \ldots, M_N$, and accessed through read or write instructions. More formally, a RAM program is defined by a "next instruction" circuit Π which, given its current state state and a value d (that will always be equal to the last-read element), outputs the next instruction and an updated state state'. Thus, if M is an array of N entries, each ν bits long, we can view an execution of a RAM program proceeding as follows. First, set $\mathsf{state} = \mathsf{start}$ and $d = 0^{\nu}$, and secondly, until termination, compute $(\mathsf{op}, l, d', \mathsf{state}') = \Pi(\mathsf{state}, d)$ and update $\mathsf{state} = \mathsf{state}'$. Then: (1) if $\mathsf{op} = \mathsf{stop}$, terminate with output d'; (2) if $\mathsf{op} = \mathsf{read}$ (and $d' = \perp$), set $d = M_l$; (3) if $\mathsf{op} = \mathsf{write}$, set $M_l = d'$ and $d = d'$.

[1] A multiplication tuple is a tuple (x, y, z) which is correct if $x \cdot y = z$, or incorrect otherwise. Here the goal of the servers is to verify that the z values given by P_S form correct tuples.

The *space complexity* of a RAM program on initial inputs x, M is the maximum number of entries used by the memory array M during the course of the execution. The *time complexity* is the number of instructions issued in the execution as described above.

In this work we focus on public-coin ZK proofs for RAM programs Π representing an NP relation $\mathcal{R}(x, M)$, where $\mathcal{R}$ and x are public and M is a large private dataset (which acts as a witness for x). In Figure 3 we describe an ideal functionality for RAM-based ZK proof, and in Section 4.3 we give a protocol realizing it.

Functionality $\mathcal{F}_{\textsf{ZK-RAM}}$

Prove: On input $(\textsf{sid}, \textsf{Prove}, \Pi, \textsf{type}, N, M)$ from $\mathcal{P}$, compute $y = \Pi(M)$. Send $(\textsf{Prove}, \Pi)$ to $\mathcal{V}$ and $\mathcal{S}$ and store (Π, y).

Verify: On input $(\textsf{sid}, \textsf{Verify}, \Pi, \textsf{type})$ from $\mathcal{V}$, query $\mathcal{S}$; if $\mathcal{S}$ returns $\textsf{fail}$, or if (Π, y) has not been stored, send $(\textsf{sid}, \Pi, 0)$ to $\mathcal{V}$. Otherwise, send $(\textsf{sid}, \Pi, y)$ to $\mathcal{V}$.

Figure 3: Ideal functionality for RAM-based ZK proofs

3 Arithmetic Circuit for ZK Verification of Array Access

In this section we construct an arithmetic circuit $C_{\textsf{check}}$ (over a binary or prime field $\mathbb{F}$) which verifies the consistency of a series of T read or write accesses to an initial array $\mathcal{M}$ of size N.

We denote by $[x]$ wire values in $C_{\textsf{check}}$ that are sensitive and should not be revealed when $C_{\textsf{check}}$ is proven in zero-knowledge. Each entry in the initial array is of the form $(i, i, \textsf{write}, [M_i])$ for $i \in [N]$, where $M = (M_1, \ldots, M_N)$ is an arbitrary initial state. Contrary to Franzese et al. [FKL+21], here our circuit assumes that the each of the first N tuples of the list $\mathcal{L}$ contains a hard-coded write operation (with unknown data values); this implies that our circuit does not need to verify that the first access to an index is always a write operation.

3.1 Constant Overhead Equality Check

Our first sub-circuit verifies the equality of two hidden values without leaking the result; this allows the equality bit to continue to be used as a hidden value within $C_{\textsf{check}}$. To obtain the result of the equality test within a hidden value, the $\textsf{EqCheck}$ sub-circuit shown in Circuit 1 makes use of an auxiliary value r which, when $x \neq y$, is set to $(x - y)^{-1}$ such that $b = (x - y) \cdot r = 1$. When $x = y$, $b = (x - y) \cdot r = 0$ for any r. The circuit then returns $1 - b$ so that 1 is output in case of equality.

Since this r requires a precise value, it must be input into the circuit using the `Input` command. However, this allows for dishonest behaviour, so the circuit must also check that: 1. r is non-zero and 2. the final result b is indeed a bit (if r was non-zero but also not equal to $(x - y)^{-1}$ when $x \neq y$, then b would not be a bit). To perform the first check, $\textsf{EqCheck}$ requires r^{-1} to be input so that $r \cdot r^{-1}$ can be verified to equal 1. For the second check, $(1 - b) \cdot b$ is tested to be equal 0, which implies $b \in \{0, 1\}$.

Zero-knowledge. If correct, both $r \cdot r^{-1}$ and $(1-b) \cdot b$ always evaluate to the same value, independently of r or b, so they can be safely checked against a constant (1 or 0) without leaking information.

Circuit 1 $\mathsf{EqCheck}([x],[y])$

1: `Input` $[r] = \begin{cases} ([x]-[y])^{-1} & \text{if } x \neq y \\ \text{random non-zero} & \text{if } x = y \end{cases}$
2: `Input` $[r^{-1}]$
3: Check that $[r] \cdot [r^{-1}]$ is equal to 1; if not, set circuit output to 0.
4: $[b] \leftarrow ([x]-[y]) \cdot [r]$
5: Check that $(1-[b]) \cdot [b]$ is equal to 0; if not, set circuit output to 0.
6: **return** $1-[b]$

Soundness. Both checks are deterministic, therefore if r and r^{-1} are incorrectly input, either of these will fail and C_{check} will output 0 with probability 1.

Cost. This circuit requires a constant number of `Input`, multiplication and constant checks (resp. 2, 3 and 2) to evaluate the equality bit of two values.

3.2 Permutation Check

Our second sub-circuit probabilistically checks that two arrays of S (tuples of) field elements are permutations of one another without revealing either the content of the arrays or the permutation that links them. We first describe the procedure in the one- and multi-dimensional case before formally presenting the $\mathsf{PermCheck}$ sub-circuit.

3.2.1 Checking a one-dimensional permutation.

We first present the checking procedure in the one-dimen-sional case, as described in [BCG+18]. Let $[A] = \begin{bmatrix}[a_1] & \cdots & [a_S]\end{bmatrix}$ and $[B] = \begin{bmatrix}[b_1] & \cdots & [b_S]\end{bmatrix}$ be two arrays in $\mathbb{F}^S$; to show that there exists a secret permutation π such that $B = \pi(A)$, we use the fact that polynomials are identical under permutation of the roots [Gro09, Nef01]. In other words, we define two polynomials $P_A, P_B \in \mathbb{F}[x]$ such that their zeros are exactly the elements of the respective arrays:

$$[P_A(x)] = \prod_{i=1}^{S}(x-[a_i]) \qquad \text{and} \qquad [P_B(x)] = \prod_{i=1}^{S}(x-[b_i]).$$

If the arrays are indeed permutations of one another, then the polynomials are defined identically and it holds that $P_A = P_B$. We check this probabilistically using the Schwartz–Zippel Lemma.

1: Receive public random challenge $r \in \mathbb{F}$.
2: Compute $r-[a_i]$ and $r-[b_i]$ for $i \in [S]$.
3: Compute the values $[P_A(r)] = \prod_{i=1}^{S}(r-[a_i])$ and $[P_B(r)]$ similarly.
4: Check that $P_A(r) - P_B(r)$ is equal to 0.

Given that P_A and P_B are both of degree S, the Schwartz–Zippel Lemma states that, if $P_A \neq P_B$, then the check in Step 4 will incorrectly pass with probability at most $S/|\mathbb{F}|$.

3.2.2 Checking a multi-dimensional permutation.

In our application, as the two lists $\mathcal{L}$ and $\mathcal{L}'$ contain tuples of 4 elements we instead need to consider matrices. However, the following analysis can be generalized to matrices of higher dimension. Let $[\mathbf{A}] = \big[[\mathbf{a}_1] \quad \cdots \quad [\mathbf{a}_S]\big]$ and $[\mathbf{B}] = \big[[\mathbf{b}_1] \quad \cdots \quad [\mathbf{b}_S]\big]$ be matrices in $\mathbb{F}^{4\times S}$; we wish to prove that the columns of $\mathbf{B}$ are a permutation of the columns of $\mathbf{A}$, i.e. there exists a permutation π such that $\mathbf{A}P_\pi = \mathbf{B}$, where P_π is the matrix permutation associated to π.

To do so, we reduce the question to the one-dimensional case using randomized inner products. First, a random challenge $\mathbf{s} \in \mathbb{F}^4$ is sampled. Then, $\mathbf{A}$ is compressed to a one-dimensional array $\mathbf{a}$ by setting $(\mathbf{a})_i = a_i = \mathbf{s} * \mathbf{a}_i$, for $i \in [S]$, where $*$ denotes the inner product of two vectors. Similarly, $\mathbf{B}$ is compressed to $\mathbf{b}$ using the same $\mathbf{s}$. (Using the same challenge $\mathbf{s}$ for all columns of both matrices is necessary since the permutation must remain secret.) Now, the procedure for the one-dimensional case presented above can be used to check that $\mathbf{b}$ is a permutation of $\mathbf{a}$.

To show that this procedure correctly checks that the columns of $\mathbf{B}$ are a permutation of the columns of $\mathbf{A}$, we show that any difference is preserved by the randomized inner product except with some probability.

Lemma 1 *Given two matrices* $\mathbf{A}, \mathbf{B}$ *as above, if there does not exist a column permutation matrix* P_π *such that* $\mathbf{A}P_\pi = \mathbf{B}$ *then the sets* $\{a_1, \dots, a_S\}$ *and* $\{b_1, \dots, b_S\}$ *are different except with probability at most* $4/|\mathbb{F}|$ *over the random choice of* $\mathbf{s} \in \mathbb{F}^4$.

Proof We can consider the linear map $f_{\mathbf{a}-\mathbf{b}}(\mathbf{s})$, defined by the matrix $\mathbf{D} = (\mathbf{A} - \mathbf{B})^T \in \mathbb{F}^{M\times 4}$. If $\mathbf{A}$ and $\mathbf{B}$ are correctly generated, $\mathbf{D} = \mathbf{0}$ and the condition $f_{\mathbf{D}}(\mathbf{s}) = 0$ holds $\forall \mathbf{s} \in \mathbb{F}^4$.
If the adversary cheated, i.e., $\mathbf{D} \neq \mathbf{0}$, we can have the following different cases:

- If only one row is incorrect, then rank $\mathbf{D} = 1$ and the rank-nullity theorem tells us dim(ker $f_{\mathbf{D}}$) = 3. This means that the probability that $\mathbf{s} \in \ker f_{\mathbf{D}}$ is $|\mathbb{F}^3|/|\mathbb{F}^4| = 1/|\mathbb{F}|$.
- If two rows are incorrect, then rank $\mathbf{D} \leq 2$. If it is 1, then we are in the same situation as before, otherwise dim(ker $f_{\mathbf{D}}$) = 2 and the probability that $\mathbf{s} \in \ker f_{\mathbf{D}}$ is $|\mathbb{F}^2|/|\mathbb{F}^4| = 1/|\mathbb{F}^2|$.
- If three rows are incorrect, then rank $\mathbf{D} \leq 3$, hence either we are in one of the situations described above or dim(ker $f_{\mathbf{D}}$) = 1 and the probability that $\mathbf{s} \in \ker f_{\mathbf{D}}$ is $|\mathbb{F}|/|\mathbb{F}^4| = 1/|\mathbb{F}^3|$.
- If we have more than 3 incorrect rows and rank $\mathbf{D} = 4$, then $f_{\mathbf{D}}$ is injective and $\ker f_{\mathbf{D}} = \{\mathbf{0}\}$. Hence, the probability of passing the test is $1/|\mathbb{F}^4|$.

Therefore the overall probability of passing the test is given $1/|\mathbb{F}| + 1/|\mathbb{F}^2| + 1/|\mathbb{F}|^3 + 1/|\mathbb{F}|^4 \leq 4/|\mathbb{F}|$.
□ This approach is similar to the one of Bootle et al. [BCG+18] which uses an inner product with a vector of powers $(1, z, z^2, \dots)$ for random value z.

3.2.3 Constructing the circuit.

We now present the PermCheck sub-circuit in Circuit 2. This takes $\nu \in \mathbb{N}$ as parameter to indicate the row-dimension of the arrays $\mathcal{L}$ and $\mathcal{L}'$; if $\nu = 4$ then we use the multi-dimensional check described above and sample a random vector $\mathbf{s} \in \mathbb{F}^\nu$. Then, the circuit performs the Schwartz–Zippel test by requiring a random $r \in \mathbb{F}$, evaluating the polynomials P_A and P_B on r and checking that they are equal, i.e. that their difference is 0.

Zero-knowledge. The only revealed information is that $P_A(r) - P_B(r)$ is equal to 0; however, this is always the case when $[\mathcal{L}']$ is a permutation of $[\mathcal{L}]$, therefore no information is leaked.

Circuit 2 $\mathsf{PermCheck}(\nu \in \{1,4\}, [\mathcal{L}], [\mathcal{L}'])$

if $\nu = 4$ **then**
  $\mathbf{s} \leftarrow \mathtt{Rand}(\mathbb{F}^{\nu})$
for $i \in [S]$ **do**
  if $\nu = 1$ **then**
    $[a_i] \leftarrow [\mathcal{L}[i]]$ and $[b_i] \leftarrow [\mathcal{L}'[i]]$
  else
    $[a_i] \leftarrow \mathbf{s} * [\mathcal{L}[i]]$ and $[b_i] \leftarrow \mathbf{s} * [\mathcal{L}'[i]]$
$r \leftarrow \mathtt{Rand}(\mathbb{F})$
$[P_A(r)] \leftarrow \prod_{i=1}^{S}(r - [a_i])$ and $[P_B(r)] \leftarrow \prod_{i=1}^{S}(r - [b_i])$
Check that $[P_A(r)] - [P_B(r)]$ is equal to 0; if not, set circuit output to 0.

Soundness. When $\nu = 1$, the one-dimensional case is sufficient to show that $\mathsf{PermCheck}$ incorrectly passes with probability at most $S/|\mathbb{F}|$. When $\nu = 4$, Lemma 1 gives us that, if $\mathcal{L}'$ is not a permutation of $\mathcal{L}$, $\{a_i\}$ and $\{b_i\}$ will be different except with probability at most $4/|\mathbb{F}|$. If the sets are different, the one-dimensional case then again implies that the last check will incorrectly pass with probability at most $S/|\mathbb{F}|$. Therefore, when $\nu = 4$, the probability that $\mathsf{PermCheck}$ incorrectly passes is at most

$$\Pr_{\mathbf{s}}[\{a_i\} = \{b_i\}] + \Pr_{\mathbf{s}}[\{a_i\} \neq \{b_i\}] \cdot \Pr_{r}[P_A(r) = P_B(r)]$$
$$\leq \frac{4}{|\mathbb{F}|} + \left(1 - \frac{4}{|\mathbb{F}|}\right)\frac{S}{|\mathbb{F}|} \leq \frac{S+4}{|\mathbb{F}|}.$$

Cost. When $\nu = 1$, this circuit requires one `Input` command, one `Rand` command, $2(S-1)$ multiplications gates and one constant equality check. When $\nu = 4$, it requires an additional ν `Rand` commands as well as $2\nu S$ multiplications to compute the inner products.

3.3 Amortized Constant Overhead Bound Test

Circuit 3 $\mathsf{BdCheck}(\{[x_i]\}_1^T, B_1, B_2)$

Arrange initial array $[\mathcal{L}] = [B_1, B_1+1, \ldots, B_2, [x_1], [x_2], \ldots, [x_T]]$ of size $S = B_2 - B_1 + 1 + T$.
$\mathtt{Input}[\mathcal{L}']$ containing entries of $\mathcal{L}$ sorted from lowest to highest.
$\mathsf{PermCheck}([\mathcal{L}], [\mathcal{L}'])$ ▷ Sets the circuit output to 0 if it fails.
for $i \in [S-1]$ **do**
  $[\alpha_i] \leftarrow \mathsf{EqCheck}([\mathcal{L}'[i]], [\mathcal{L}'[i+1]])$
  $[\beta_i] \leftarrow [\mathcal{L}'[i+1]] - [\mathcal{L}'[i]]$
  Check that $[\alpha_i] + [\beta_i]$ is equal to 1; if not, set circuit output to 0.
Check that $[\mathcal{L}'[1]] = B_1$ and that $[\mathcal{L}'[S]] = B_2$; if not set circuit output to 0.

Our third sub-circuit $\mathsf{BdCheck}$, shown in Circuit 3, verifies in zero-knowledge that a set $\{[x_i]\}$ of T values are all contained within specified public bounds B_1 and B_2.

To do so, it first creates an array $\mathcal{L}$ of all values from B_1 to B_2, both included, and then appends all T values to be checked, forming an array of size $S = B_2 - B_1 + 1 + T$. Using `Input` commands, it then requires an array $[\mathcal{L}']$ of same size S which is expected to be an ordered permutation of $\mathcal{L}$. (Even though the values $B_1, \ldots, B_2$ were not hidden in $\mathcal{L}$, all of the values of $\mathcal{L}'$ must now remain hidden so that no information is leaked about $\{[x_i]\}$.) By verifying that the first entry of $\mathcal{L}'$ is equal to B_1 and the last entry of $\mathcal{L}'$ is equal to B_2, the circuit verifies that $B_1 \leq x_i \leq B_2$ for all $i \in [T]$.

As in the circuit for equality checking, the `Input` commands allow for dishonest behaviour so several properties of $\mathcal{L}'$ must additionally be checked. First, BdCheck calls PermCheck to verify that $\mathcal{L}'$ is indeed a permutation of $\mathcal{L}$ and therefore that no value has been modified.

Second, the circuit checks that successive entries in $\mathcal{L}'$ are either equal to each other or differ by exactly 1. In a correctly input $\mathcal{L}'$, this is always the case as every value $[x_i]$ should be equal to one value between B_1 and B_2.

It does so by first computing $\alpha_i = \mathsf{EqCheck}(\mathcal{L}'[i], \mathcal{L}'[i+1])$ and then $\beta_i = \mathcal{L}'[i+1] - \mathcal{L}'[i]$. Note that if $\mathcal{L}'[i+1] = \mathcal{L}'[i]$, then $\alpha_i + \beta_i = 1 + 0 = 1$, if instead $\mathcal{L}'[i+1] = \mathcal{L}'[i]+1$ then $\alpha_i + \beta_i = 0+1 = 1$. If however $\mathcal{L}'[i+1] \notin \{\mathcal{L}'[i], \mathcal{L}'[i]+1\}$, then $\alpha_i + \beta_i \neq 1$ and therefore checking for this equality is sufficient to complete this second check.

Finally, BdCheck verifies that $\mathcal{L}'[1] = B_1$ and that $\mathcal{L}'[S] = B_2$. This, combined with the second check, implies that $B_1 \leq x_i \leq B_2$ for all $i \in [T]$.

Zero-knowledge. First, PermCheck guarantees zero-knowledge of $[\mathcal{L}']$ during the first check. Next, if $[\mathcal{L}']$ was input correctly, then $[\alpha_i] + [\beta_i]$ should always equal 1 and therefore no information is leaked by checking this. Finally, given that B_1 and B_2 are public values and included in $[\mathcal{L}]$, checking the first and last entry of $[\mathcal{L}']$ does not reveal any information on any $[x_i]$ if $[\mathcal{L}']$ was input correctly.

Soundness. The checks on $[\alpha_i] + [\beta_i]$ and the first and last entries of $[\mathcal{L}']$ are all deterministic, so BdCheck makes C_{check} output 0 with probability 1 if any of these fail. PermCheck is probabilistic in nature, however, so BdCheck has the same soundness error overall, i.e. $S/|\mathbb{F}|$ since $\mathcal{L}$ is one-dimensional here.

Cost. This circuit amortizes the cost of checking whether $B_1 \leq x \leq B_2$ by checking T values at the same time. This requires S calls to `Input`, one PermCheck call, $S-1$ EqCheck calls and $S+2$ constant equality checks.

3.4 Putting everything together

We now present the complete C_{check} circuit which verifies the consistency of accesses, held as tuples $([l], t, [op], [d])$ within the list $\mathcal{L}$. Recall that it does so by requiring a second list $\mathcal{L}'$ to be an ordering of $\mathcal{L}$ and by verifying that (1) $\mathcal{L}'$ is a permutation of $\mathcal{L}$; (2) $\mathcal{L}'$ is correctly ordered, first according to l and then according to t for entries concerning the same address; (3) all addresses are within bounds; (4) all operations are either reads or writes; and (5) all read tuples contain the same value as the last one written to the same address.

Checking (1) and (3). The first is done by calling $\mathsf{PermCheck}(4, [\mathcal{L}], [\mathcal{L}'])$ and the second is done by checking that $[l'_{N+T}] = N$.

Checking (2). Here we check equalities, which is done using EqCheck, but also the inequalities $t'_i < t'_{i+1}$, in the case where $l'_i = l'_{i+1}$. Since the t_i values are public within $\mathcal{L}$, and we know that $\mathcal{L}'$ is a permutation of $\mathcal{L}$, it holds that $1 \leq [t'_i] \leq N+T$ for all $i \in [N+T]$. Letting $[\tau_i] = [t_{i+1}] - [t_i]$, we see that $0 < [\tau_i] \implies [t_i] < [t_{i+1}]$. Therefore, calling $\mathsf{BdCheck}([\tau_i], 1, N+T-1)$ would allow to test this (setting 1 as the lower bound ensures the strict inequality; setting $N+T-1$ as the upper bound ensures all values of τ_i are included). However, if successive tuples access different addresses, then successive values of t are not ordered in this way; e.g. with the tuples $(1, 2, *, *)$ and $(2, 1, *, *)$. Therefore calculating τ_i in this manner does not yield the correct check.

To fix this, we include only the τ_i values for accesses to the same address, i.e. those for which the equality $l'_i = l'_{i+1}$ holds. Setting $[\alpha_i] \leftarrow \mathsf{EqCheck}([l'_i], [l'_{i+1}])$, we can instead let $[\tau_i] \leftarrow [\alpha_i]([t'_{i+1}] - [t'_i]) + (1 - [\alpha_i])$. The first summand includes $[t'_{i+1}] - [t'_i]$ when the equality holds, and nullifies it otherwise, and the second summand ensures $\tau_i > 0$ when the equality does not hold. Now, $\mathsf{BdCheck}(\{\tau_i\}, 1, N + T - 1)$ will pass exactly when the t values are correctly ordered within groups of accesses to the same address l.

To finally check the ordering of the addresses, similarly to the $\mathsf{BdCheck}$ circuit, verifying that $[l'_{i+1}] = [l'_i] + 1$ when $[l'_{i+1}] \neq [l'_i] + 1$ does not require a second $\mathsf{EqCheck}$. Instead we compute $[\lambda_i] \leftarrow [l'_{i+1}] - [l'_i]$ and check that $[\alpha_i] + [\lambda_i]$ is equal to 1. If $[l'_{i+1}] \notin \{[l'_i], [l'_i] + 1\}$, this will not pass.

Checking (4). For every $i \in [N + T]$, as $[op'_i]$ should be a bit, representing either read or write, we check that $(1 - [\mathsf{op}'_i])[\mathsf{op}'_i] = 0$.

Checking (5). We check that adjacent tuples contain either (a) different addresses, (b) equal memory values, or (c) a write operation. As $[\alpha_i]$ already contains the equality bit of the two addresses, and (2) checked that addresses either are equal or differ by one, then $1 - [\alpha_i]$ is exactly the truth value required for (a). For (b) we set $[\beta_i] \leftarrow \mathsf{EqCheck}([d'_i], [d'_{i+1}])$. Finally for (c), $[op'_{i+1}]$ is its own equality bit with respect to the write operation. To evaluate $(a) \vee (b) \vee (c)$, we then compute $\neg(\neg(a) \wedge \neg(b) \wedge \neg(c))$:

$$[\gamma_i] \leftarrow 1 - [\alpha_i] \cdot (1 - [\beta_i]) \cdot (1 - [\mathsf{op}'_{i+1}]),$$

and check that $[\gamma_i]$ is equal to 1 for every $i \in [N + T - 1]$.

Circuit 4 $C_{\mathsf{check}}([\mathcal{L}])$

Assume initial array is of the form

$$[\mathcal{L}] = [(1, 1, \mathsf{write}, [M_1]), \ldots, (N, N, \mathsf{write}, [M_N]), \ldots$$
$$\ldots ([\ell_{N+1}], N + 1, [\mathsf{op}_{N+1}], [d_{N+1}]), \ldots, ([\ell_{N+T}], N + T, [\mathsf{op}_{N+T}], [d_{N+T}])]$$

$\mathtt{Input}[\mathcal{L}']$ containing entries of $\mathcal{L}$ sorted first by ℓ then by t.
$\mathsf{PermCheck}(4, [\mathcal{L}], [\mathcal{L}'])$
for $i \in [N + T - 1]$ **do**
    Set $[\alpha_i] \leftarrow \mathsf{EqCheck}([\ell'_i], [\ell'_{i+1}])$
    Set $[\lambda_i] \leftarrow [\ell'_{i+1}] - [\ell'_i]$
    Set $[\tau_i] \leftarrow [\alpha_i] \cdot ([t'_{i+1}] - [t'_i]) + (1 - [\alpha_i])$
    Check that $[\alpha_i] + [\lambda_i]$ is equal to 1; if not, set circuit output to 0.
    Check that $(1 - [\mathsf{op}'_i]) \cdot [\mathsf{op}'_i]$ is equal to 0; if not, set circuit output to 0.
    $[\beta_i] \leftarrow \mathsf{EqCheck}([d'_i], [d'_{i+1}])$
    Set $[\gamma_i] \leftarrow 1 - [\alpha_i] \cdot (1 - [\beta_i]) \cdot (1 - [\mathsf{op}'_{i+1}])$
    Check $[\gamma_i]$ is equal to 1; if not, set circuit output to 0.
$\mathsf{BdCheck}(\{[\tau_i]\}_{i=1}^{N+T-1}, 1, N + T - 1)$
Check $[\ell'_{N+T}]$ is equal to N; if not, set circuit output to 0.
If circuit output was not set to 0 at any point, output 1.

The C_{check} circuit. The final circuit is presented in Circuit 4; it performs checks (1) through (5) as described above and, if the output was never set to 0 by a failed constant check, then it outputs 1 to signify that all accesses contained in $\mathcal{L}$ are consistent with the initial memory and one another.

Correctness. We first note that, if all `Input` gates are given correctly, then the C_{check} circuit will always output 1, independently of the output of the `Rand` gates.

Zero-knowledge. The zero-knowledge properties of the EqCheck, PermCheck and BdCheck sub-circuits was argued in previous sections. As for the C_{check} circuit, the check of step 8 is always equal to 1 if $\mathcal{L}'$ was input correctly, and so is the check of step 12, therefore no information is leaked by either. Similarly, $[\mathsf{op}'_i]$ should always be a bit, so step 9 also does not leak information. Finally, N is already publicly contained in $\mathcal{L}$ as the address of the last tuple, so step 14 does not reveal information either if $\mathcal{L}'$ was input correctly.

Soundness. PermCheck is the only non-deterministic check performed in the circuit, at steps 3 and 13 (within BdCheck). We therefore have the following.

Lemma 2 *If $[\mathcal{L}]$ or $[\mathcal{L}']$ is incorrectly input at step 2 of C_{check} and step 2 of* BdCheck *respectively, then C_{check} will output 0 with probability at most $3(N+T)^2/|\mathbb{F}|^2$.*

Proof The first check is a permutation of tuples of 4 elements, therefore the analysis of PermCheck gives us that it can fail (i.e. not output 0 even though $[\mathcal{L}']$ is not a permutation of $[\mathcal{L}]$) with probability at most $(N+T+4)/|\mathbb{F}|$. The second check, within BdCheck, uses the one-dimensional permutation check and can therefore fail with probability at most $2(N+T-1)/|\mathbb{F}|$.

The output of C_{check} can only be 1 if neither check succeeds in catching the incorrect input. Since they concern different input values, and are performed with independent randomness, the probability that both fail is exactly the product of the probabilities that each fail, which is less than $3(N+T)^2/|\mathbb{F}|^2$ (assuming $N+T \geq 6$). □

Cost. Since PermCheck and BdCheck are called outside of the **for** loop, the execution of C_{check} costs $O(N+T)$ standard arithmetic operations with $O(N+T)$ additional inputs.

4 Zero-Knowledge Proof of Array Access

The standard (stateless) zero-knowledge proof functionality for Boolean or arithmetic circuits, described in Figure 1 is only suitable for deterministic circuits. As described in Section 3, our C_{check} circuit makes use of `Rand` gates to probabilistically verify the consistency of the access list. To ensure soundness, this requires that the verification randomness be generated only *after* the inputs have been committed to, as otherwise the prover could use the randomness to commit to incorrect inputs which would nonetheless satisfy the checks.

In this section, we first introduce an "input" extension of the $\mathcal{F}_{\mathsf{ZK}}$ functionality which then accepts circuits to be evaluated both on stored and fresh input values. Alongside, we also present the version of $\mathcal{F}_{\mathsf{ZKArray}}$ that our initial protocol realizes and discuss the differences with the version of Franzese et al. Then we present Π_{ZKArray}, our zero-knowledge protocol for private read/write array access, which realizes our $\mathcal{F}_{\mathsf{ZKArray}}$ using the extended zero-knowledge functionality, and prove its security in the UC framework. Finally, we discuss how our $\mathcal{F}_{\mathsf{ZKArray}}$ and Π_{ZKArray} can both be extended to provide stateless proofs for richer circuits that include both arithmetic operations and array accesses.

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

The functionality runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary $\mathcal{S}$.
It is parametrized by $\mathsf{type} = \{\mathsf{Boolean}, \mathsf{Arithmetic}\}$.

Init: On input $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, \mathcal{L})$ from $\mathcal{P}$, if no previous initialization command has been given, and if $\mathcal{L}$ matches type, store type and $\mathcal{L}$ and send $(\mathsf{sid}, \mathsf{Initialized}, \mathcal{P})$ to $\mathcal{V}$ and $\mathcal{S}$. Otherwise, ignore this command.

Input: On input $(\mathsf{sid}, \mathsf{Input}, v)$ from $\mathcal{P}$, append v to $\mathcal{L}$ if the type of v matches type and send $(\mathsf{sid}, \mathsf{Input})$ to $\mathcal{V}$. If Init has not been given, or if Prove has already been given, ignore this command instead.

Prove: Receive $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C, x)$ from the prover. If Init has not been given, if the type of C or x does not match type, or if $(C, *)$ is already stored, ignore this command. Otherwise, compute $y = C(\mathcal{L}, x)$, send (Prove, C) to $\mathcal{S}$ and $\mathcal{V}$, and store (C, y).

Verify: On input $(\mathsf{sid}, \mathsf{Verify}, C)$ from $\mathcal{V}$, query $\mathcal{S}$. If $\mathcal{S}$ sends fail, or if (C, y) is not stored, send $(\mathsf{sid}, C, 0)$ to $\mathcal{V}$. Otherwise, send (sid, C, y) to $\mathcal{V}$.

Figure 4: Ideal functionality for circuit-based ZK proof with separate input command.

4.1 $\mathcal{F}_{\mathsf{ZKin}}$ and $\mathcal{F}_{\mathsf{ZKArray}}$ Functionalities

Figure 4 describes the $\mathcal{F}_{\mathsf{ZKin}}$ functionality for (stateless) zero-knowledge proof of Boolean or arithmetic circuit with a separate **Input** command. This functionality must be initialized once with sid and $\mathsf{type} \in \{\mathsf{Boolean}, \mathsf{Arithmetic}\}$. Since the aim is to allow for inputs to be given ahead of the **Prove** command, the initialization also accepts a list $\mathcal{L}$ of values, whose type must match type, which the functionality then stores. Afterwards, the **Input** command may be called several times to append values v to the initial list $\mathcal{L}$; the verifier $\mathcal{V}$ is informed of each of these calls.

The **Prove** command may then be called once, during which $\mathcal{P}$ specifies the circuit C and any additional input x. The functionality then evaluates C jointly on $\mathcal{L}$ and x, stores the result, and informs $\mathcal{S}$ and $\mathcal{V}$.

Finally, the verifier may call the **Verify** command, specifying the circuit C; this ensures that $\mathcal{P}$ and $\mathcal{V}$ agree on the circuit that should be proven. If $C(\mathcal{L}, x)$ has not been proven by $\mathcal{P}$, or if $\mathcal{S}$ decides to interrupt, then $\mathcal{F}_{\mathsf{ZKin}}$ informs $\mathcal{V}$ of the failure and stops. Otherwise, it sends $(\mathsf{sid}, C, 1)$ to $\mathcal{V}$ and stops.

Figure 5 presents a *stateless* version of the $\mathcal{F}_{\mathsf{ZKArray}}$ functionality. As opposed to the stateful one presented by Franzese et al. [FKL+21], this functionality does not extend $\mathcal{F}_{\mathsf{ZKin}}$, and therefore does not have a **Prove** command for arbitrary circuits, but only provides commands to initialize and access a memory array in zero-knowledge and also check the consistency of the accesses that were made. We discuss the extension of our $\mathcal{F}_{\mathsf{ZKArray}}$ functionality with a **Prove** command in Section 4.3.

4.2 ZKArray Protocol

We provide a protocol for private read/write array access, which first allows the prover $\mathcal{P}$ to commit to an array of values, and then to read or write values from or to the committed data structure in such a way that the verifier $\mathcal{V}$ does not learn the address being accessed, nor the operation being performed or the value being written.

Our protocol Π_{ZKArray}, described in Figure 6, makes use of C_{check} presented in Section 3 to

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

The functionality runs with $\mathcal{P}, \mathcal{V}$ and an adversary $\mathcal{S}$.
PARAMETERS: The functionality is parametrized by a flag $\mathsf{f} \in \{0,1\}$, the size N of the array, and an upper bound T on the number of memory accesses.

Init: On input $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, M, N, T)$ from $\mathcal{P}$, if no previous initialization command has been given, and if the entries of M match type, store type and M; send $(\mathsf{sid}, \mathsf{Initialized}, \mathsf{type}, \mathcal{P}, N, T)$ to $\mathcal{V}$ and $\mathcal{S}$. Set $\mathsf{f} = 1$. Otherwise, ignore this command.

Access: On input $(\mathsf{sid}, \mathsf{Access}, l, \mathsf{op}, d)$ from $\mathcal{P}$, if $l \geq N$ set $\mathsf{f} = 0$, otherwise:

- if $\mathsf{op} = \mathsf{Read}$: if $M_l \neq d$ then set $\mathsf{f} = 0$;
- if $\mathsf{op} = \mathsf{Write}$: set $M_l = d$.

In all cases, send $(\mathsf{sid}, \mathsf{Access})$ to $\mathcal{V}$ and $\mathcal{S}$.

Check: Upon receiving $(\mathsf{sid}, \mathsf{Check}, T)$ from $\mathcal{V}$, query $\mathcal{S}$. If $\mathcal{S}$ sends fail, return $(\mathsf{sid}, 0)$ to $\mathcal{V}$; otherwise, when $\mathcal{S}$ sends $\mathsf{Deliver}$, if $\mathsf{f} = 0$ then send $(\mathsf{sid}, 0)$ to $\mathcal{V}$, otherwise send $(\mathsf{sid}, 1)$ and halt.

Figure 5: Functionality for *stateless* ZK proof for private read/write array access

Protocol Π_{ZKArray}

PARAMETERS: N is the size of the array, and T an upper bound on the number of accesses.

Init: On input a memory array M with contents $M_1, \ldots, M_N$, and a type, $\mathcal{P}$ creates a list $\mathcal{M} = [(1, 1, \mathsf{write}, M_1), \ldots, (N, N, \mathsf{write}, M_N)]$, initializes a counter $t = N$ and creates two empty lists $\mathcal{L}, \mathsf{AuxIn}$. It then sends $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, \mathcal{M})$ to $\mathcal{F}_{\mathsf{ZKin}}$.

Access: On input (l, op, d), $\mathcal{P}$ increments t and appends (l, t, op, d) to $\mathcal{L}$.

Check: $\mathcal{P}$ and $\mathcal{V}$ perform the following steps:

1. $\mathcal{P}$ parses $C_{\mathsf{check}}(\mathcal{M}||\mathcal{L})$ and for each $\mathtt{Input}(x)$ command it appends x to AuxIn.
2. $\mathcal{P}$ sends $(\mathsf{sid}, \mathsf{Input}, \mathcal{L}||\mathsf{AuxIn})$ to $\mathcal{F}_{\mathsf{ZKin}}$ which then sends $(\mathsf{sid}, \mathsf{Input})$ to $\mathcal{V}$.
3. $\mathcal{V}$ sends r_i to $\mathcal{P}$ for each $\mathtt{Rand}$ command in C_{check}.
4. $\mathcal{P}$ sends $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C^{\{r_i\}}_{\mathsf{check}}, \emptyset)$ to $\mathcal{F}_{\mathsf{ZKin}}$.
5. $\mathcal{V}$ sends $(\mathsf{sid}, \mathsf{Verify}, C^{\{r_i\}}_{\mathsf{check}})$ to $\mathcal{F}_{\mathsf{ZK}}$. It returns whatever $\mathcal{F}_{\mathsf{ZKin}}$ returns and stops.

Figure 6: Protocol realizing $\mathcal{F}_{\mathsf{ZKArray}}$ in the $\mathcal{F}_{\mathsf{ZKin}}$-hybrid model.

realize $\mathcal{F}_{\mathsf{ZKArray}}$. At **Init**, the prover receives the initial memory array $M = [M_1, \ldots, M_N]$. From it, it creates a list of initial access tuples $\mathcal{M}[i] = (i, i, \mathsf{write}, M_i)$ which enforces that every address is written to according to the entry in the array and that the first N memory accesses are write operations. After initializing the access counter at $t = N$, ready to be incremented, and creating an empty list $\mathcal{L}$ to contain the future accesses, $\mathcal{P}$ commits to the initial memory by sending $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, \mathcal{M})$ to $\mathcal{F}_{\mathsf{ZKin}}$. Afterwards, for each **Access** operation and its corresponding (ℓ, op, d) input, $\mathcal{P}$ increments t and appends $(\ell, t, \mathsf{op}, d)$ to the list $\mathcal{L}$.

When T access operations have been completed, the **Check** procedure begins. First, $\mathcal{P}$ parses C_{check} for $\mathtt{Input}$ gates and computes the required value for each, appending it to AuxIn each time. Note that no such auxiliary input within C_{check} is dependent on the output of a $\mathtt{Rand}$ gate, therefore all values can be computed by $\mathcal{P}$ before receving the outputs for the `Rand` gates. After parsing all `Input` gates, $\mathcal{P}$ commits to these values by sending $(\mathsf{sid}, \mathsf{Input}, \mathcal{L}||\mathsf{AuxIn})$ to $\mathcal{F}_{\mathsf{ZKin}}$.

The verifier receives confirmation of the commitment from the functionality and proceeds to sampling a random value r_i for each `Rand` within C_{check} before sending all of them to $\mathcal{P}$.

Now, both $\mathcal{P}$ and $\mathcal{V}$ can replace the output of the `Rand` gates by the values sampled above to specify the circuit to a deterministic one, which we label $C^{\{r_i\}}_{\mathsf{check}}$. Finally, it is this circuit, without additional input, that $\mathcal{P}$ proves with $\mathcal{F}_{\mathsf{ZKin}}$ and that $\mathcal{V}$ asks to verify.

Theorem 2 *Protocol Π_{ZKArray} securely realizes $\mathcal{F}_{\mathsf{ZKArray}}$ in the $\mathcal{F}_{\mathsf{ZKin}}$-hybrid model with statistical error at most $3(N+T)^2/|\mathbb{F}|^2$.*

Proof We describe a simulator $\mathcal{S}$ interacting with $\mathcal{P}$ and $\mathcal{V}$ and internally simulating the $\mathcal{F}_{\mathsf{ZKin}}$ functionality. Figure 7 presents $\mathcal{S}$ when $\mathcal{P}$ is honest; Figure 8 presents the case when $\mathcal{P}$ is malicious.

Simulator $\mathcal{S}$ for honest prover

$\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{ZKin}}$ honestly internally, and $\mathcal{P}$ as follows:

Init: Upon receiving $(\mathsf{sid}, \mathsf{Initialized}, \mathsf{type}, \mathcal{P}, N, T)$ from $\mathcal{F}_{\mathsf{ZKArray}}$, set $M_i = 0$ for $i \in [N]$, generate $\mathcal{M}$ from $M = (M_i)_i$ as in Π_{ZKArray}, initialize $t = N$ and $\mathcal{L}, \mathsf{AuxIn} = \emptyset$, and send $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, \mathcal{M})$ to $\mathcal{F}_{\mathsf{ZKin}}$.

Access: Upon receiving $(\mathsf{sid}, \mathsf{Access})$ from $\mathcal{F}_{\mathsf{ZKArray}}$, increment t, set $l = 1$, $\mathsf{op} = \mathsf{Read}$ and $d = 0$ and append (l, t, op, d) to $\mathcal{L}$.

After all T access tuples were sent, if $\mathcal{V}$ is honest, send $(\mathsf{sid}, \mathsf{Check}, T)$ to $\mathcal{F}_{\mathsf{ZKArray}}$.

Check: When queried by $\mathcal{F}_{\mathsf{ZKArray}}$ on whether to deliver, simulate the checking protocol as follows.

1. Parse $C_{\mathsf{check}}(\mathcal{M}||\mathcal{L})$, honestly compute each `Input` gate and append to AuxIn.
2. Send $(\mathsf{sid}, \mathsf{Input}, \mathcal{L}||\mathsf{AuxIn})$ to $\mathcal{F}_{\mathsf{ZKin}}$.
3. Receive $\{r_i\}$ from $\mathcal{A}$ if $\mathcal{V}$ is malicious; otherwise generate $\{r_i\}$ honestly and add them to transcript visible by $\mathcal{A}$.
4. Send $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C^{\{r_i\}}_{\mathsf{check}}, \emptyset)$ to $\mathcal{F}_{\mathsf{ZKin}}$, which then sends $(\mathsf{Prove}, C^{\{r_i\}}_{\mathsf{check}})$ to $\mathcal{A}$.
5. If $\mathcal{V}$ is honest, send $(\mathsf{sid}, \mathsf{Verify}, C^{\{r_i\}}_{\mathsf{check}})$ to $\mathcal{F}_{\mathsf{ZKin}}$.
6. If $\mathcal{A}$ instructs $\mathcal{F}_{\mathsf{ZKin}}$ to fail, then send fail to $\mathcal{F}_{\mathsf{ZKArray}}$. Otherwise, send $\mathsf{Deliver}$ to $\mathcal{F}_{\mathsf{ZKArray}}$.

Figure 7: Simulator for an honest prover for Π_{ZKArray} in the $\mathcal{F}_{\mathsf{ZKin}}$-hybrid model.

Honest prover. If $\mathcal{V}$ is also honest, then $\mathcal{S}$ in Figure 7 generates dummy values as inputs for $\mathcal{P}$ and samples the $\{r_i\}$ values honestly. Because (1) nothing is sent to $\mathcal{V}$ directly, (2) $\mathcal{F}_{\mathsf{ZKin}}$ guarantees the zero-knowledge proof of C_{check}, and (3) C_{check} does not leak information along the checks that it performs, then the transcript seen by $\mathcal{A}$ is indistinguishable from a real execution of Π_{ZKArray}.

If $\mathcal{V}$ is malicious, then the only degree of freedom that the adversary $\mathcal{A}$ possesses in Π_{ZKArray} is to sample the r_i values according to a different distribution. However, the correctness of C_{check} ensures that, with honest behaviour from $\mathcal{P}$, the output is always 1, independently of the r_i values, therefore the output of $\mathcal{F}_{\mathsf{ZKArray}}$ is identically distributed.

Malicious prover. In Figure 8, $\mathcal{S}$ receives the inputs of $\mathcal{P}^*$ via $\mathcal{F}_{\mathsf{ZKin}}$ and forwards them to $\mathcal{F}_{\mathsf{ZKArray}}$

Simulator $\mathcal{S}$ for malicious prover

$\mathcal{S}$ processes queries to $\mathcal{F}_{\mathsf{ZKin}}$ as follows:

Init: Upon receiving $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, \mathcal{M})$ from $\mathcal{P}^*$, extract $M = (M_1, \dots, M_N)$ from $\mathcal{M}$ and send $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, M, N, T)$ to $\mathcal{F}_{\mathsf{ZKArray}}$. Then send $(\mathsf{sid}, \mathsf{Initialized}, \mathcal{P})$ to $\mathcal{V}$ and $\mathcal{A}$.

Access: Upon receiving $(\mathsf{sid}, \mathsf{Input}, \mathcal{L})$ from $\mathcal{P}^*$, extract the first T access tuples (l, t, op, d), for $t \in [N+1, T]$, from $\mathcal{L}$ and, for each tuple, send $(\mathsf{sid}, \mathsf{Access}, l, \mathsf{op}, d)$ to $\mathcal{F}_{\mathsf{ZKArray}}$. Then send $(\mathsf{sid}, \mathsf{Input})$ to $\mathcal{V}$.

After all T access tuples were sent, if $\mathcal{V}$ is honest, send $(\mathsf{sid}, \mathsf{Check}, T)$ to $\mathcal{F}_{\mathsf{ZKArray}}$.

Check: When queried by $\mathcal{F}_{\mathsf{ZKArray}}$ on whether to deliver, simulate the rest of the checking protocol as follows.

1. If $\mathcal{V}$ is honest, generate $\{r_i\}$ honestly and send them $\mathcal{A}$.
2. Upon receiving $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C_{\mathsf{check}}^{\{r_i\}}, \emptyset)$ from $\mathcal{P}^*$, process it honestly and send $(\mathcal{P}, C_{\mathsf{check}}^{\{r_i\}})$ to $\mathcal{A}$ if $y = 1$.
3. If $\mathcal{V}$ is honest, send $(\mathsf{sid}, \mathsf{Verify}, C_{\mathsf{check}}^{\{r_i\}})$ to $\mathcal{F}_{\mathsf{ZKin}}$.
4. If $\mathcal{A}$ instructs $\mathcal{F}_{\mathsf{ZKin}}$ to fail, then send fail to $\mathcal{F}_{\mathsf{ZKArray}}$. Otherwise, send $\mathsf{Deliver}$ to $\mathcal{F}_{\mathsf{ZKArray}}$.

Figure 8: Simulator for malicious prover for Π_{ZKArray} in the $\mathcal{F}_{\mathsf{ZKin}}$-hybrid model.

appropriately, both for the initial memory and the T access tuples. If $\mathcal{P}^*$ cheated for any of these, then $\mathcal{F}_{\mathsf{ZKArray}}$ will set $\mathsf{f} = 0$ internally.

$\mathcal{S}$ instructs $\mathcal{F}_{\mathsf{ZKArray}}$ to fail only when $\mathcal{A}$ instructs $\mathcal{F}_{\mathsf{ZKin}}$ to do so, therefore the output given to $\mathcal{V}$ by $\mathcal{F}_{\mathsf{ZKArray}}$ will be identically distributed to a real execution except when the probabilistic check of C_{check} incorrectly outputs 1. By Lemma 2, this happens with probability at most $3(N+T)^2/|\mathbb{F}|^2$. □

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

Here we show how to extend $\mathcal{F}_{\mathsf{ZKArray}}$ to be able to describe a protocol for RAM-based computation and implement the ideal functionality $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$ given in Figure 3.

To accept richer circuits, constructed from both arithmetic or Boolean operations and array accesses, we modify our functionality and protocol as follows.

Functionality. To extend our $\mathcal{F}_{\mathsf{ZKArray}}$ with a **Prove**(C) command, we merge the **Access** commands into the computation of C. That is, when given (C, x) from **Prove** and M from **Init**, the extended funtionality $\mathcal{F}_{\mathsf{ZKArray}}^{\mathsf{ex}}$ computes $C(M, x)$ and, every time an `Access` is encountered within C, it queries $\mathcal{P}$ to input (l, op, d) as the access operation.

The corresponding **Verify** command then supersedes **Check** and performs the following operations. As **Check**, it first of all verifies that all the accesses given by $\mathcal{P}$ are consistent with the initial M and with each other. Additionally, it also verifies that the accesses given by $\mathcal{P}$ are consistent with C; i.e. that $\mathcal{P}$ provided the correct l, op and d that C instructed to perform at that moment. Finally, as for $\mathcal{F}_{\mathsf{ZKin}}$, it stores the result $y = C(M, x)$ in order to validate, or not, the successful computation of C.

Protocol. To extend Π_{ZKArray} to handle richer circuits, we expand the circuit that $\mathcal{P}$ submits to $\mathcal{F}_{\mathsf{ZKin}}$. Namely, $\mathcal{P}$ constructs the same list $\mathcal{L}$ of access tuples and, in addition to $C_{\mathsf{check}}^{\{r_i\}}$, also proves (1) the arithmetic or Boolean circuits which output the tuples that C is expecting and (2) C as a whole, simplified to a purely arithmetic or Boolean circuit by using the tuples in $\mathcal{L}$ as constant wire values.

Realizing $\mathcal{F}_{\mathsf{ZK\text{-}RAM}}$ in the $\mathcal{F}^{\mathsf{ex}}_{\mathsf{ZKArray}}$-hybrid model. Given the command $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, \Pi, \mathsf{type}, N, M)$, $\mathcal{P}$ first executes Π until termination to calculate the number T of accesses required; it then sends $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, M, N, T)$ to $\mathcal{F}_{\mathsf{ZKArray}}$. It then sends $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C_\Pi, \emptyset)$ to $\mathcal{F}^{\mathsf{ex}}_{\mathsf{ZKArray}}$ where C_Π is the circuit built as a succession of next-instruction circuits $\Pi(\mathsf{state}, d)$ interleaved with `Access` instructions.

5 Realizing $\mathcal{F}_{\mathsf{ZKin}}$ with Limbo

In this section, we show how the ZK proof system Limbo [DOT21] can be generalized to securely realize $\mathcal{F}_{\mathsf{ZKin}}$ in the $\mathcal{F}_{\mathsf{Commit}}$-hybrid model.

Handling Init and Input commands. Recall that the MPC protocol used by Limbo is divided into two phases; a first where P_S sends the inputs of the circuit and the outputs of the multiplication gates to the computation servers, and a second where servers, using $\mathcal{F}_{\mathsf{Rand}}$ and helped by P_S, execute the MultCheck protocol and send the output to P_R.

To realize $\mathcal{F}_{\mathsf{ZKin}}$, we let the Limbo prover $\mathcal{P}$ perform the following before beginning the first phase. When the **Init** command is given, $\mathcal{P}$ commits to $\mathcal{L}$ as the beginning of the input, and waits. For every **Input** command given afterwards, $\mathcal{P}$ commits to v and appends it to $\mathcal{L}$ and waits further. When the **Prove** command is given, $\mathcal{P}$ appends x to $\mathcal{L}$ and considers this final $\mathcal{L}$ as the input to the circuit C given by **Prove**.

Equality with constant checks. We note that the Limbo ZK-IOP system handles addition gates for free and uses the MultCheck protocol to handle multiplication gates. Since the C_{check} of Section 3 also makes use of equality checks against contants, we quickly present how Limbo handles such circuit elements. In the general case, to check that $[x]$ is equal to a constant c, the Limbo protocol can add the multiplication tuple $(1, [x], c)$ to the list of tuples to check. Since 1 and c are constants here, no help is required from P_S which implies that no extra communication is required.

In the specific case that the result of a multiplication is checked against a constant, i.e. verifying that $[x] \cdot [y] = c$, then the tuple $([x], [y], c)$ can be added to the list. This also implies no extra communication since the result of the multiplication is a public value and does not need to be given by P_S. This is useful for checking bits, for example, where it must be verified that $[1-b] \cdot [b]$ is equal to 0.

UC Security in the $\mathcal{F}_{\mathsf{Commit}}$-Hybrid Model. In Figure 9 we present the $\mathsf{Limbo}_{\mathsf{UC}}$ protocol, the generalized version of Limbo described above which we also rephrase for the UC framework. For the detailed description of the protocol and the relevant definitions, we refer the reader to [DOT21].

Theorem 3 *Protocol* $\mathsf{Limbo}_{\mathsf{UC}}$ *presented in Figure 9 UC-realizes* $\mathcal{F}_{\mathsf{ZKin}}$ *in the* $\mathcal{F}_{\mathsf{Commit}}$*-hybrid model, for semi-honest verifiers with perfect secrecy and for malicious provers with statistical error:*

$$\epsilon = \frac{1}{n} + \delta\left(1 - \frac{1}{n}\right),$$

Protocol $\mathsf{Limbo_{UC}}$

PARAMETERS: a ρ-phase MPC protocol in the client/server model which computes arithmetic or Boolean circuits and is $(P_R, n-1)$-private and $(P_S, 0)$-robust.

Init: On input $(\mathsf{sid}, \mathsf{Init}, \mathsf{type}, \mathcal{L})$, $\mathcal{P}$ prepares to run the MPC protocol in its head. It samples r_S and $\{r_i\}_{i\in[n]}$ and, as the sender client P_S, uses these to secret-share $\langle\mathcal{L}\rangle$ accross the different servers P_i. It then sends $(\mathsf{sid}, \mathsf{Commit}, \langle\mathcal{L}\rangle_i)$ to $\mathcal{F}_{\mathsf{Commit}}$.

Input: On input $(\mathsf{sid}, \mathsf{Input}, v)$, $\mathcal{P}$ continues to act as P_S by appending v to $\mathcal{L}$, secret-sharing $\langle v\rangle$, and sending $(\mathsf{sid}, \mathsf{Commit}, \langle v\rangle_i)$ to $\mathcal{F}_{\mathsf{Commit}}$.

Prove: On input $(\mathsf{sid}, \mathsf{Prove}, \mathcal{P}, \mathcal{V}, C, w)$, $\mathcal{P}$ appends w to $\mathcal{L}$, secret-shares $\langle w\rangle$ and invokes P_S on input $(\mathcal{L}; r_S)$ and each P_i on input $(\langle\mathcal{L}\rangle_i, \langle w\rangle_i; r_i)$. This computes the views $(\mathsf{view}_1^1, \ldots, \mathsf{view}_n^1)$ of the servers in phase 1 with which $\mathcal{P}$ sends $(\mathsf{sid}, \mathsf{Commit}, \mathsf{view}_i^1)$ to $\mathcal{F}_{\mathsf{Commit}}$. Then, for $j \in [2, \rho]$,

- $\mathcal{V}$ sends a random challenge R_{j-1} to $\mathcal{P}$.
- $\mathcal{P}$ continues the protocol: it invokes P_S and each P_i on input R_{j-1} which computes the views $(\mathsf{view}_1^j, \ldots, \mathsf{view}_n^j)$. $\mathcal{P}$ then sends $(\mathsf{sid}, \mathsf{Commit}, \mathsf{view}_i^j)$ to $\mathcal{F}_{\mathsf{Commit}}$.

Verify: After commiting to the last view of the servers, the following takes place.

- $\mathcal{V}$ sends a final random challenge R_ρ to $\mathcal{P}$.
- $\mathcal{P}$ invokes P_S and each P_i on input R_ρ; this computes the final view view_R which $\mathcal{P}$ sends to $\mathcal{V}$.
- $\mathcal{V}$ outputs $(\mathsf{sid}, C, 0)$ if P_R rejects the execution; otherwise, $\mathcal{V}$ samples a subset $V \subset [n]$ of server views that it wishes to check and sends V to $\mathcal{P}$.
- Together, $\mathcal{P}$ and $\mathcal{V}$ open the commitments within $\mathcal{F}_{\mathsf{Commit}}$ for all the views specified by V.
- $\mathcal{V}$ outputs $(\mathsf{sid}, C, 0)$ if the server views are inconsistent with each other or with view_R; otherwise, $\mathcal{V}$ outputs $(\mathsf{sid}, C, 1)$.

Figure 9: The generalized version of Limbo, presented for the UC framework.

where δ is the $(P_S, 0)$-robustness error of the MPC protocol.

Proof For semi-honest verifiers, the security of $\mathsf{Limbo_{UC}}$ follows from the $(P_R, n-1)$-privacy of the MPC protocol. Since the simulator internally emulates $\mathcal{F}_{\mathsf{Commit}}$, it can generate dummy values of behalf of the honest prover, of whose inputs it has no knowledge, and simulate the openings according to the privacy simulator for the MPC protocol.

For malicious provers, the security of $\mathsf{Limbo_{UC}}$ follows from the same argument as Limbo's [DOT21, Theorem 3.4]. Indeed, the addition of $\mathcal{F}_{\mathsf{Commit}}$ and the separation of the commitments to the inputs of C do not increase the cheating strategies for $\mathcal{P}$. Therefore, $\mathcal{V}$ outputs 0 exactly when $\mathcal{F}_{\mathsf{ZKin}}$ would, except with the same proability as Limbo: $\epsilon = \frac{1}{n} + \delta\left(1 - \frac{1}{n}\right)$, where δ is the $(P_S, 0)$-robustness error of the MPC protocol taken over the random challenges sent by $\mathcal{V}$. □

As described in [DOT21, Section 3.3], the soundness error can be further reduced by increasing the number of MPC instances computed in parallel. The same can be applied here to give a reduced soundness error

$$\epsilon_\tau = \frac{1}{n^\tau} + \delta\left(1 - \frac{1}{n^\tau}\right).$$

Non-Interactive Proof. As our Π_{ZKArray} protocol is entirely stateless and uses only public randomness, it can be compiled to a non-interactive (NI) proof in the $\mathcal{F}_{\mathsf{ZKin}}$-hybrid model according to the Fiat–Shamir transform.

Furthermore, the Limbo protocol for $\mathcal{F}_{\mathsf{ZKin}}$ can itself be compiled to an NI proof with the same methodology. However, due to its high number of interaction rounds between the prover and the verifier, the soundness analysis of the resulting protocol is non-trivial [DOT21, Section 6].

Similarly, our generalized $\mathsf{Limbo}_{\mathsf{UC}}$ protocol can be transformed to an NI proof where each call to $\mathcal{F}_{\mathsf{Commit}}$ is replaced by calls to a random oracle that generates randomness in place of $\mathcal{V}$. Combined with an NI version of Π_{ZKArray}, the random oracle would then generate the randomness for the Rand gates on behalf of $\mathcal{V}$ between the Input and Prove calls to $\mathcal{F}_{\mathsf{ZKin}}$. We leave the exact soundness analysis and parameter generation to further work.

6 Implementation Results

We implemented our protocol in C++, using a slightly modified version of Limbo as described in Figure 9. We first added support for arbitrary fields on top of the implementation for binary fields of [DOT21]; and we then built on the Bristol Format of circuits by adding Access gates.

The circuit is represented as a text file which specifies the size of the memory that will be needed, as well as the number of input wires, output wires and hardcoded wires. For each hardcoded wire, the file also specify their value. Finally, the file also consists of a list of gates in topological order where each gate is specified as the operation that it performs and the wires it operates on.

We start by parsing the circuit in order to propagate hardcoded wires. For example, an addition of two hardcoded wires is also an harcoded wire. Then, we transform every Access gate into a set of new input wires which will define the lists $\mathcal{L}$ and $\mathcal{L}'$. Once the whole circuit has been analyzed, we build C_{check} following the procedure described in Circuit 4 using the newly defined wires. At this point, we have a circuit composed only of *standard* arithmetic gates which Limbo can evaluate.

As an additional improvement with respect to the original code, we also support Rand gates. These gates are implicitly used in our protocol every time the Verifier needs to send a challenge to the Prover in the PermCheck circuit. However, if the need arises for a specific use case, our implementation can handle such Rand gates within the circuit itself, thus giving more freedom for future implementation of statisitcal check in the spirit of PermCheck.

Finally we also propose a multi-threaded implementation, where each repetition of the proof is run on its own thread. As for the original Limbo, this trivial parallelization does not allow us to divide the running time by the number of threads, because there are some places where threads have to join; but it nonetheless gives a significant improvement.

6.1 Performance

All the benchmarks were done on a desktop computer with an Intel i9-9900 (3.1GHz) CPU and 128GB of RAM. We only provide proving times and proof size; and do not take into account communication time between parties.

In all cases, we show the running time of our implementation using $\mathbb{F} = GF(2^{61} - 1)$ averaged over 20 runs for varying RAM sizes.

In Figure 10 we show figures for the initialization phase of the array for three parameter sets to emphasize potential trade-off between running time and proof size as well as the benefit of multi-threading; all sets provide statistical security of 40 bits for interactive proofs. In the case of multi-threading, we chose to run with 8 parties and 14 repetitions because we had 14 threads available on our CPU.

We observe that the initialization phase of the array costs an amortized 22 `Mult` gates per memory slot. Subsequent accesses, with sensitive operation and memory location cost an amortized 24 `Mult` gates per access. In terms of concrete performance, when focusing on better runtime for a single thread, each access amounts to roughly 0.4ms and 1.8KB. For the multi-threaded case, each access costs 0.12ms and 1.8KB.

In Table 1 we summarize our comparison with other work. We compare our results to Franzese et al. [FKL+21], noting that their performance is measured for proofs using rings of 32-bit integers, whereas our implementation uses $GF(2^{61}-1)$ which is 30 bits larger. On a single thread, using parameters optimizing for running time, we are about 40 times slower with proof sizes 60 times bigger; if we instead trade-off running time for better proof size, we are about 110 times slower with proof sizes 30 times bigger.

We also compare our work to the BubbleRAM and the more recent PrORAM protocols [HK20, HK21], both are tailored specifically for private-coin protocols in the prime field setting. For a RAM size of 2^{18} elements in $GF(2^{40}-87)$ BubbleRAM (resp. PrORAM) achieves an amortized access time of 0.15ms (resp. 0.01ms) and communication size of 1.5KB (resp. 0.4KB). Therefore, while providing memory elements that are 21 bits larger (about 1.5x), our protocol is only 3 times slower with 1.2 times bigger proof size than BubbleRAM and 40 times slower with 5 times bigger proof size than PrORAM. In light of these comparisons, we emphasize that MPCitH protocols are designed to be public-coin and are therefore inherently slower and with bigger proof sizes than protocols that can take advantage of private coins.

Finally we remark that MPCitH protocols can be significantly speed up using a multi-threaded implementation. If hardware allows, we can run each repetition of the proof on a separate thread. We show here that with 14 threads we can match the running time of BubbleRAM with a proof size only 1.2 times bigger.

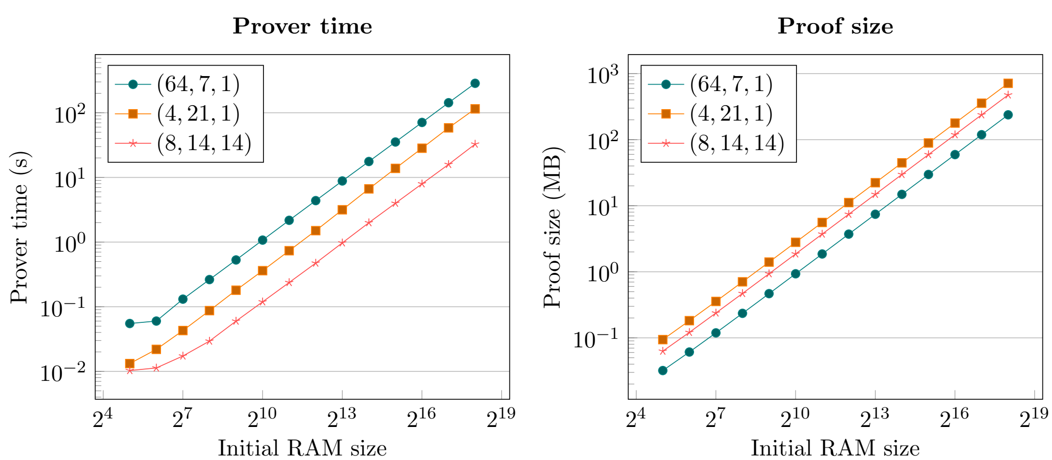

Figure 10: Prover time and proof size in the interactive case for initialization of different sizes of RAM. We specify (#parties, #repetitions, #threads).

[1]Access Time and Access Size are considered for a RAM of size 2^{18} elements.

Scheme	Algebraic Structure	Asymptotic Complexity	Access Time (ms)	Access Size (KB)
BubbleRAM [HK20][1]	$GF(2^{40}-87)$	$O(log^2(N))$	0.15	1.5
PrORAM [HK21][1]	$GF(2^{40}-87)$	$O(log(N))$	0.01	0.4
Franzese et al. [FKL+21]	$\mathbb{Z}_{2^{32}}$	$O(1)$	0.01	0.031
Ours $(64,7,1)$	$GF(2^{61}-1)$	$O(1)$	1.11	0.920
Ours $(4,21,1)$	$GF(2^{61}-1)$	$O(1)$	0.42	2.82
Ours $(8,14,1)$	$GF(2^{61}-1)$	$O(1)$	0.44	1.82
Ours $(8,14,14)$	$GF(2^{61}-1)$	$O(1)$	0.12	1.82

Table 1: Comparison of our protocol with previous work in the designated-verifier setting. For our scheme we specify (#parties, #repetitions, #threads).

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.

[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+18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 595–626. Springer, Heidelberg, December 2018.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.

[BCTV14] 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.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for arithmetic circuits with nested disjunctions. In *Advances in Cryptology, Crypto 2021*, 2021. `https://eprint.iacr.org/2020/1410`.

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

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications, 2021.

[DOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge mpcith-based arguments, 2021. https://ia.cr/2021/215.

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for ram programs. Cryptology ePrint Archive, Report 2021/979, 2021. https://ia.cr/2021/979.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. https://ia.cr/2021/1063.

[Gro09] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 192–208. Springer, Heidelberg, August 2009.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[HK20] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 2055–2074. ACM Press, November 2020.

[HK21] David Heath and Vladimir Kolesnikov. Proram: Fast $o(\log n)$ private coin zk oram. Cryptology ePrint Archive, Report 2021/587, 2021. https://eprint.iacr.org/2021/587.

[HMR15] Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 150–169. Springer, Heidelberg, August 2015.

[HYD+21] David Heath, Yibin Yang, David Devecsery, Vladimir Kolesnikov, and Marco Guarnieri. Zero-knowledge for everything and everyone: Fast zk processor with cached oram for ansi c programs. IEEE Symp. on Security and Privacy, 2021.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[MRS17] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 501–531. Springer, Heidelberg, April / May 2017.

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 116–125. ACM Press, November 2001.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits, 2021.

[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, 2021.

[XZZ+19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, August 2019.

[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 *ACM Conf. on Computer and Communication Security (CCS)*. ACM, 2021.