PrORAM
Fast $O(\log n)$ Private Coin ZK ORAM

David Heath and Vladimir Kolesnikov

Georgia Institute of Technology, Atlanta, GA, USA
{heath.davidanthony,kolesnikov}@gatech.edu

Abstract. We construct a *concretely efficient* Zero Knowledge (ZK) Oblivious RAM (ORAM) that consumes $2 \log n$ oblivious transfers (OTs) of length-2σ secrets per access of an arithmetic value, for statistical security parameter σ and array size n. This is an asymptotic and concrete improvement over previous best (concretely efficient) ZK ORAM BubbleRAM of Heath and Kolesnikov ([HK20a], CCS 2020), whose access cost is $\frac{1}{2} \log^2 n$ OTs of length-2σ secrets.

ZK ORAM is essential for proving statements that are best expressed as RAM programs, rather than Boolean or arithmetic circuits.

Our construction is private-coin ZK. We integrate it with [HK20a]'s ZK Proof (ZKP) protocol and prove the resulting ZKP system secure.

We implemented PrORAM in `C++`. Compared to the state-of-the-art BubbleRAM, our PrORAM is $\approx 10\times$ faster for arrays of size 2^{20} of 40-bit values.

Keywords: Oblivious RAM, Zero Knowledge

1 Introduction

Zero Knowledge (ZK) proofs (ZKP) allow an untrusted prover $\mathcal{P}$ to convince an untrusted verifier $\mathcal{V}$ of the truth of a given statement *while revealing nothing additional*. ZKPs are foundational cryptographic objects useful in many contexts. Early ZK focused on proofs of specific statements, but more recent systems handle *arbitrary* statements, so long as the statements are encoded as circuits.

Motivation. Unfortunately, many statements are difficult to encode as circuits and are more easily and much more efficiently expressed as RAM machine programs; indeed, most standard algorithms are formalized for RAM machines.[1] Importantly, recent work, e.g. [HK20a], demonstrates that support for writing proof statements as arbitrary `C` programs is within reach. ORAM is a major cost factor in [HK20a]'s ZK virtual machine, responsible for 1/3 to 1/2 or more of the total cost, since ORAM is accessed at each CPU step. An efficient ZK ORAM

[1] While RAM machines reduce to circuits, improving the reduction will allow more efficient proofs.

would dramatically improve the performance of (already practical) ZK virtual machine of [HK20a].

Most ORAM research targets either (1) an untrusted server holding a client's private data or (2) the secure multiparty computation setting. ZK ORAMs have been less studied. ZK, as compared to these more explored settings, gives a crucial advantage: $\mathcal{P}$ can precompute the order in which the proof circuit will access each RAM element. Prior work [HK20a] has shown that this knowledge suffices to build a circuit-based ORAM that incurs only $\frac{1}{2}\log^2 n$ oblivious transfers (OTs) per access. While the constant factor of this approach is excellent, the $\log^2$ scaling can be costly for large RAMs.

In this work, we construct an efficient ZK ORAM that we call PrORAM. PrORAM consumes only $2\log n$ OTs per access.

1.1 High level intuition of our approach

$\mathcal{P}$ and $\mathcal{V}$ evaluate the proof circuit or program by jointly processing it gate-by-gate. The validity of the proof is ensured by the fact that each circuit wire holds an *authenticated secret share* that $\mathcal{P}$ cannot forge.

Our prover $\mathcal{P}$ stores the RAM locally on her system, but the authenticated contents are masked by one-time-pad masks chosen by $\mathcal{V}$. Because $\mathcal{P}$ stores the RAM locally and because she knows the RAM access order, she can directly access each requested index. From here, the crucial problem is that each RAM slot is masked by a distinct value chosen by $\mathcal{V}$. To ensure $\mathcal{V}$, who does not know the access order, can authenticate a value read from the RAM, the value must have a mask that is *independent of the accessed index.* Thus, ORAM essentially reduces to 'aligning' masks without leaking the RAM access order to $\mathcal{V}$. We arrange mask alignment by allowing $\mathcal{P}$ to authentically and obliviously permute $\mathcal{V}$'s chosen masks into a desired order.

For a RAM with n slots, a single permutation on $2n$ elements suffices to support the next n accesses. Using a Waksman permutation network, this permutation can be achieved by $2n\log n$ OTs. Thus, each access consumes amortized $2\log n$ OTs.

1.2 Contribution

We construct a private-coin ZK ORAM, PrORAM, that uses only $2\log n$ OTs per access, while previous ZK ORAM has cost $1/2\log^2 n$. We instantiate our ORAM in the [JKO13] ZK framework, resulting in a ZK protocol with 2 rounds (4 flows) of communication when using standard OT, such as [KOS15].[2]

- We present PrORAM in technical detail, and prove it correct and secure.

[2] In our implementation, we use Ferret OT [YWL+20], which greatly improves communication. Ferret processes OTs in very large chunks, requiring additional rounds for each next chunk. This round complexity increase is small and contributes little to total runtime. E.g., in concrete terms, two added rounds give $\approx 2^{23}$ OTs.

– We integrate PrORAM into the arithmetic ZK protocol of [HK20a]. Thus, our construction allows proofs of arbitrary arithmetic statements encoded as circuits with access to a highly efficient RAM. Note, [HK20a]'s ZK virtual machine is a circuit; our ORAM can be directly plugged in their ZK VM.

  We formalize the resulting construction in the [JKO13] garbled-circuit based ZK proof framework and prove the system correct and secure.
– We implemented our approach in `C++` and explore its concrete performance. As compared to BubbleRAM [HK20a], a state-of-the-art ORAM for the same setting, and for size 2^{20} RAMs, PrORAM improves communication by $> 4\times$ and runs $> 10\times$ faster on a commodity laptop. Our more significant computation improvement follows from the fact that our algorithms are friendlier to cache than BubbleRAM's (see Section 9).

2 Related Work

Our contribution is an efficient ORAM for an interactive Zero Knowledge protocol. In our review of related work, we discuss both ZK protocols and ORAMs.

Zero Knowledge. ZK proofs [GMR85,GMW91] are fundamental cryptographic primitives. ZK proofs of knowledge (ZKPoKs) [GMR85] allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that she holds an input i to a circuit or program $\mathcal{C}$ such that $\mathcal{C}(i) = 1$. Early ZK protocols focused on specific relations, but more recently research attention has shifted to proofs of arbitrary statements encoded as circuits. Our work seeks to augment this more recent line with efficient RAM access so that proof statements can be expressed as programs for RAM machines.

Interactive ZK. The most closely related works allow an interacting $\mathcal{P}$ to quickly convince $\mathcal{V}$ that she holds a satisfying assignment to a circuit [JKO13,FNO15,HK20c,HK20a,WYKW20]. This line of work is attractive because it features costs that scale linearly in the circuit size with low constants. Thus, if $\mathcal{P}$ and $\mathcal{V}$ wish to finish a proof as fast as possible, these constructions are excellent choices.

[JKO13] was the first work to construct concretely efficient proofs of arbitrary circuits by reducing ZKPs to *garbled circuits* (GCs). Recently, new works further improved this style of interactive ZK. [HK20c] improved GC-based ZK by reducing the communication consumption of conditional branching. [HK20a] and [WYKW20] both introduced GC-like constructions that allow proofs over arithmetic circuits. This improvement from Boolean to arithmetic dramatically improves performance.

We embed our ZK ORAM in [HK20a]'s arithmetic protocol. Their protocol, which is based entirely on oblivious transfer (OT), is made extremely efficient by recent innovations in OT extension [BCG$^+$19,YWL$^+$20]. By embedding into their protocol, we provide arbitrary arithmetic circuits that can access an efficient RAM. We review their protocol detail in Section 4.1.

Succinct and non-interactive ZK (NIZK). While our focus is on the interactive setting, we review NIZK as well to put our work in perspective. Ishai et al. [IKOS07], introduced the 'MPC-in-the-head' paradigm: here, $\mathcal{P}$ emulates in her head a multiparty computation protocol that evaluates the proof statement amongst virtual players. $\mathcal{V}$ challenges $\mathcal{P}$ to open random portions of the evaluation transcript; if these portions are consistent with the MPC protocol, $\mathcal{V}$ gains confidence that the prover has a witness. By allowing $\mathcal{V}$ to inspect transcripts of only some virtual players, the protocol protects $\mathcal{P}$'s input. MPC-in-the-head is the backbone of numerous ZK techniques [CDG+17,KKW18,AHIV17,BFH+20].

Succinct non-interactive arguments of knowledge (SNARK) techniques construct small proofs with fast verification time, e.g. [BCG+13,CFH+15]. Early SNARKs required a semi-trusted party; more recent works introduced STARKs (succinct transparent arguments of knowledge) [BBHR18]. STARKs do not require trusted setup and rely on more efficient primitives.

While these works have excellent ZK performance in many settings (e.g., small proof size, fast verification time, non-interactivity), they do not (yet) achieve low-constant linear scaling of interactive proofs. For example, the virtual machine implemented in [BCG+13] runs at 1Hz vs the 2.1KHz clockrate achieved in [HK20a]. This motivates our focus on concretely fast ORAM for the faster interactive setting.

Oblivious RAM. While our focus is on ZK ORAM, we include in our review the broader area of general ORAM as well. Oblivious RAM (ORAM) [GO96] is a significant area of cryptographic research, e.g. [AKL+20,SvS+13]. ORAM allows array access while hiding access patterns. ZK ORAM permits the prover to know the access pattern, and, of course, can be instantiated with a general-purpose ORAM. However, ZK ORAM is a simpler object, and could be constructed more efficiently. Such higher-efficiency ZK ORAMs are relatively unexplored.

Some prior ZK works that interface with RAM use standard oblivious RAM (ORAM) as a black box, e.g. [MRS17,HMR15]. Two general purpose ORAMs stand out in terms of concrete performance. 'Floram' scales well to large memory sizes [Ds17]. [RS19] is a square-root ORAM preferable for smaller memory sizes.

A small number of works have explored ZK ORAM. BubbleRAM [HK20a], our main point of comparison, is a ZK-specific ORAM with excellent concrete performance. BubbleRAM is integrated in [HK20a]'s arithmetic ZK protocol, which we also use as a wrapper for our ZK ORAM. BubbleRAM consumes only $1/2 \log^2 n$ oblivious transfers of length-2σ secrets (σ is a statistical security parameter) per access to a size-n RAM. We note that, although not discussed in [HK20a], the BubbleRAM construction is just a circuit; it can be instantiated as a public-coin ZKP protocol, and thus can be used in non-interactive ZK, e.g., via Fiat-Shamir. Like our approach, their technique relies on the oblivious permutation of values.

[BCG+13] integrate a ZK ORAM construction [BCGT13] in their ZK proof system. Their technique has $O(\log^2 T)$ cost per access, where T is the number of steps in the program. They only implement their ORAM as a zkSNARK, and achieve poor concrete costs, compared to our work. We do not compare our performance to [BCG+13] in detail.

Prior general ORAM constructions (e.g., [AKL+20]) achieve the optimal $O(\log n)$ complexity. However, the hidden constants are high, and concretely-best constructions have much higher asymptotic complexity, as high as $O(\sqrt{n})$ [Ds17,RS19]. In addition to better complexity, our constants are also lower. This is not surprising, as we solve a simpler problem.

In sum, we do not further compare our work to general-purpose ORAM, and focus on fast ZK ORAM. Here the recent BubbleRAM [HK20a] is state-of-the-art both asymptotically and concretely. We focus our comparison on BubbleRAM (see Section 9).

3 Notation

- $\mathcal{P}$ is the prover. We refer to $\mathcal{P}$ by she, her, hers, etc.
- $\mathcal{V}$ is the verifier. We refer to $\mathcal{V}$ by he, him, his, etc.
- σ is the statistical security parameter (e.g., 40).
- κ is the computational security parameter (e.g., 128).
- $x \in_{\$} S$ denotes that the value x is drawn uniformly from the set S.
- $\langle x, y \rangle$ denotes a pair of values where x is held by $\mathcal{V}$ and y is held by $\mathcal{P}$.
- We write $a \triangleq b$ to denote that a is *defined* to be b.
- p denotes a prime integer.
- $\mathbb{Z}_p$ denotes the field of integers $\{0..p-1\}$.
- $\mathbb{Z}_p^\times$ denotes the set of integers $\{1..p-1\}$.
- We work with *authenticated sharings* of values held between $\mathcal{V}$ and $\mathcal{P}$. The authentic sharing of a value $x \in \mathbb{Z}_p$ is denoted by $[\![x]\!]$. We define authentic sharings and an algebra over such sharings in Section 4.1. A sharing consists of two *shares*, one held by $\mathcal{V}$ and one by $\mathcal{P}$.
- Authenticated sharings use uniform masks chosen by $\mathcal{V}$. It is sometimes convenient to make this mask explicit. $[\![x]\!]_M$ is an authenticated share of x that uses the mask M (see Section 4.3).
- We also work with standard *additive sharings*. We denote the additive sharing of a value $x \in \mathbb{Z}_p$ by $(\!|x|\!)$. Additive sharings are discussed in Section 4.4.
- We view RAMs as *arrays* of values, and hence work extensively with arrays:
  - In general we use capital variables to denote arrays, e.g. A.
  - When clear from context, n denotes the number of array slots. When needed for precision, we use $|A|$ to denote the number of array slots in A.
  - We consider arrays where each array *slot* may hold more than one integral value. When clear from context, s denotes the *slot size*, i.e., the number of integer values stored in each array slot.
    Flexibly sized array slots both allow arrays of complex objects and also are crucial for preventing $\mathcal{P}$ from accessing an arbitrary RAM slot rather than the program-dictated slot: we store an explicit RAM index in each slot and perform an equality check as part of the ZK proof.
  - The set $(\mathbb{Z}_p^s)^n$ denotes the prime field arrays of n slots each with size s.
  - $A[i]$ denotes the value stored in the ith slot of A. We use zero-based indexing.
  - $A[i := x]$ denotes an array update. The expression $A[i := x]$ is a new array whose contents are identical to A except that slot i is set to x. This notation *does not* denote a program statement that mutates an array in computer memory, but rather denotes the construction of a fresh mathematical object.

- When clear from context, we extend notation over field elements to arrays. For example, if A and B are two arrays of field elements with matching length and slot size, $A + B$ denotes the array containing the pointwise addition of the contents. We similarly extend share notation to arrays, $[\![A]\!]$ denotes an array where each element is an authentic sharing. We also extend array access notation: $[\![A[i]]\!]$ is the sharing of the ith element of array A.
- If $i \leq j$, then $A[i..j]$ denotes the *subarray* of elements $A[i]..A[j-1]$. The subarray does not include the jth element. We write $A[i..]$ to denote the subarray starting from index i and containing all subsequent elements of A.
- $[\cdot]$ denotes the empty array. $[a]$ denotes an array holding only the value a.
- We sometimes *concatenate* arrays. $A \mid B$ is the composite array containing each element of A followed by each element of B.

– We work with permutations that map points in time to array locations being accessed. We represent such permutations by arrays over the natural numbers such that for a given permutation π, $\pi[t] = i$ indicates that location i is accessed at time t.

– It will be convenient to keep track of a complementary view of the access order that we refer to as a *timetable*. A timetable $\mathcal{T}$ is an array over the natural numbers such that $\mathcal{T}[i] = t$ indicates that location i was last written at time t. In general, a timetable is not a permutation.

4 Preliminaries

In this section, we present technical background to our work needed to understand our contribution. In particular, we review [HK20a]'s arithmetic ZK protocol and discuss permutation networks.

4.1 Authenticated Share Algebra

We now review authenticated secret shares and the operations they support, where the output of each operation *is also authenticated*. Our ORAM is built on this share algebra.We use [HK20a]'s efficient arithmetic protocol, where the parties operate over shares using a combination of local operations and OT. Crucially, although the parties compute using OT, each of $\mathcal{P}$'s OT inputs can be precomputed from her proof witness. Thus, all OTs can be executed in parallel, and the resulting protocol runs in constant rounds.

Authenticated Shares. In the protocol, $\mathcal{P}$ and $\mathcal{V}$ hold *authenticated sharings* of values in a field $\mathbb{Z}_p$ for a σ-bit prime p (our implementation instantiates p as $2^{40} - 87$, the largest 40 bit prime). An authenticated sharing consists of two *shares*, one held by $\mathcal{V}$ and one by $\mathcal{P}$. We denote a sharing where $\mathcal{V}$'s share is $s \in \mathbb{Z}_p$ and $\mathcal{P}$'s share is $t \in \mathbb{Z}_p$ by writing $\langle s, t\rangle$. At the start of the protocol, $\mathcal{V}$ samples a non-zero global value $\Delta \in_\$ \mathbb{Z}_p^\times$. Consider a sharing $\langle X, x\Delta - X\rangle$ where $X \in \mathbb{Z}_p$ is chosen by $\mathcal{V}$. A sharing of this form is a *valid* sharing of the semantic value $x \in \mathbb{Z}_p$. We use the shorthand $[\![x]\!]$ to denote a valid sharing:

$$[\![x]\!] \triangleq \langle X, x\Delta - X\rangle$$

Sharings have two key properties:

1. $\mathcal{V}$'s share gives no information about the semantic value. This holds trivially: $\mathcal{V}$'s share is independent of x.
2. $\mathcal{P}$'s share is 'unforgeable': $\mathcal{P}$ cannot use $x\Delta - X$ to construct $y\Delta - X$ for $y \neq x$. We ensure this by hiding from $\mathcal{P}$ both the additive mask X and the authentication value Δ. This, combined with the fact that (1) the multiples of Δ are uniformly distributed over the field, and (2) the chosen prime p is large enough to achieve our desired security ensures that $\mathcal{P}$ can forge $[\![y]\!]$ only by guessing $y\Delta - X$, which only succeeds with probability $\frac{1}{p-1}$.

Opening shares. $\mathcal{P}$ must, at distinguished parts of the circuit, open her shares to $\mathcal{V}$. Let $[\![x]\!]$ be a valid authenticated sharing. When the two parties agree to open a share, we require that $\mathcal{V}$ knows the expected value x. This information is dictated by the circuit; thus $\mathcal{P}$ opening a share to $\mathcal{V}$ proves that the share represents a specific constant value. To complete the opening, $\mathcal{P}$ sends her share $x\Delta - X$ to $\mathcal{V}$, and $\mathcal{V}$ checks that the share is indeed valid (recall, $\mathcal{V}$ knows Δ and X). For complex proofs, $\mathcal{P}$ might open *many* shares to $\mathcal{V}$. Thus, [HK20a] adds a simple optimization: rather than sending each share separately, $\mathcal{P}$ instead accumulates a hash digest of all opened shares and sends this to $\mathcal{V}$. $\mathcal{V}$ can locally reconstruct the same hash and check that the two are equal. Thus, $\mathcal{P}$ sends only κ bits to open an arbitrary number of sharings.

Linear Operations. [HK20a] induced a *vector space* structure over authenticated sharings where sharings are vectors and publicly agreed constants are scalars. The vector space operations (addition, subtraction, and scaling by public constants) allow the parties to locally perform linear operations over sharings:

– To compute an authenticated sharing of a sum of shares, parties locally add their respective shares:

$$\begin{aligned}[\![x]\!] + [\![y]\!] &= \langle X, x\Delta - X\rangle + \langle Y, y\Delta - Y\rangle \\ &\triangleq \langle X + Y, (x + y)\Delta - (X + Y)\rangle = [\![x + y]\!]\end{aligned}$$

  To authentically subtract sharings, parties subtract their respective shares.
– To authentically scale a sharing by a public constant, the parties locally multiply their respective shares by the constant:

$$c[\![x]\!] = c\langle X, x\Delta - X\rangle \triangleq \langle cX, cx\Delta - cX\rangle = [\![cx]\!]$$

The parties also have access to a unit vector:

$$[\![1]\!] \triangleq \langle \Delta, 0\rangle$$

Here, the sharing mask X is $0 - \Delta$. Note that the mask X is not known to $\mathcal{P}$ because $\mathcal{P}$ does not know Δ. With this unit vector, the parties can locally construct authenticated sharings of arbitrary publicly agreed values.

Vector-Scalar Multiplication. It is not sufficient to only consider linear operations. We also need a form of non-linear operation; we use a form of vector-scalar multiplication where the scalar is known to be in $\{0, 1\}$, but is unknown to

$\mathcal{V}$. (Vector-scalar multiplication where $\mathcal{P}$ chooses scalar $a \in \mathbb{Z}_p$ can be achieved by $\lceil \log p \rceil$ applications of this special form.)

Let $x \in \{0, 1\}$ be held by $\mathcal{P}$ and let $y_1, ..., y_n \in \mathbb{Z}_p$ be a vector of field elements. Let the parties hold sharings $[\![y_1]\!], ..., [\![y_n]\!]$ and suppose they wish to compute $[\![xy_1]\!], ..., [\![xy_n]\!]$ (while $\mathcal{P}$'s input x is not authenticated, it could be verified later by an appropriately applied opening). First, $\mathcal{P}$ locally multiplies her shares by x. Thus the parties together hold:

$$\langle Y_1, xy_1\Delta - xY_1\rangle, ..., \langle Y_n, xy_n\Delta - xY_n\rangle$$

These intermediate sharings are invalid: the shares in the ith sharing do not sum to $y_i\Delta$. To resolve this, the parties participate in a single 1-out-of-2 OT where $\mathcal{V}$ acts as the sender. $\mathcal{V}$ uniformly draws n values $Y'_i \in_\$ \mathbb{Z}_p$ and allows $\mathcal{P}$ to choose between the following two vectors:

$$Y'_1, ..., Y'_n \qquad Y'_1 - Y_1, ..., Y'_n - Y_n \tag{1}$$

$\mathcal{P}$ chooses based on x and receives as output the vector $Y'_1 - xY_1, ..., Y'_n - xY_n$. The parties can now compute a valid authenticated sharing for each vector index:

$$\langle Y'_i, xy_i\Delta - xY_i - (Y'_i - xY_i)\rangle = \langle Y'_i, xy_i\Delta - Y'_i\rangle = [\![xy_i]\!]$$

A vector-scalar multiplication of a length n vector requires a 1-out-of-2 OT of $n\lceil \log p \rceil$-bit secrets. In practice, we instantiate multiplication with the Ferret OT technique [YWL$^+$20].

4.2 Implementing Standard Circuit Gates

Typical circuits include multiplication gates, not special vector-scalar gates where $\mathcal{P}$ chooses the scalar, as described above. There is a simple reduction from standard multiplication gates to [HK20a]'s vector-scalar multiplication gates and *opening gates* (an opening gate on input $[\![x]\!]$ simply requires $\mathcal{P}$ to open her share to $\mathcal{V}$, see Section 4.1): To authentically compute $[\![ab]\!]$ from inputs $[\![a]\!]$ and $[\![b]\!]$, instead compute $a'[\![1, b]\!] \mapsto [\![a', a'b]\!]$ by vector-scalar multiplication where $\mathcal{P}$ chooses a' freely, and then check that the $[\![a - a']\!] = [\![0]\!]$ using an opening gate. This check forces $\mathcal{P}$ to choose $a' = a$, and prevents her from multiplying incorrectly. We choose to keep vector-scalar gates and opening gates because these gates are highly efficient and because this reduction is simple. Each standard multiplication gate uses one vector-scalar gate and one opening gate.

Vector-scalar gates also allow $\mathcal{P}$ to provide input bits. To input $\mathcal{P}$'s private bit x, the parties compute $x[\![1]\!] = [\![x]\!]$ using a vector-scalar gate.

Other standard gates, e.g. addition and subtraction, are directly handled by the construction and do not require opening gates.

4.3 Explicit-Mask Sharings

Section 4.1 introduced an algebra over authenticated sharings. In the algebra as presented so far, we consider tuples of the form $\langle X, x\Delta - X\rangle$ where $X \in_\$ \mathbb{Z}_p$ is a uniform mask. For the purposes of our construction, it will be convenient to also consider sharings that use a *specific* mask chosen by $\mathcal{V}$. Thus, we introduce new notation for a sharing masked by a particular value:

$$[\![x]\!]_M \triangleq \langle M, x\Delta - M\rangle$$

That is, $[\![x]\!]_M$ is a sharing of x where the parties use the *specific* mask M, rather than an arbitrary mask.

We extend this notation to arrays: if A, B are equal-length arrays of $\mathbb{Z}_p$ elements, then $[\![A]\!]_B$ denotes an authentic share of A where each mask is in B:

$$[\![A[i]]\!]_B = \langle B[i], A[i]\Delta - B[i]\rangle$$

For convenience, we extend this notation so that we can mask a short array by a long array: the above array notation holds even if B is longer than A.

4.4 Standard Additive Sharings

Our construction relies on the parties' ability to manipulate secret masks chosen by $\mathcal{V}$ and unknown to $\mathcal{P}$. The algebra presented in Section 4.1 is not suitable, because it only supports sharings where $\mathcal{P}$ knows in cleartext each semantic value. We therefore also consider more traditional additive secret shares where neither party knows the underlying value.

Let $x \in \mathbb{Z}_p$ be an arbitrary value. In an additive share of x, $\mathcal{V}$ holds a uniform mask $M \in \mathbb{Z}_p$ and $\mathcal{P}$ holds $x - M$: together the parties hold $\langle x, x - M\rangle$. We use the shorthand $(\!|x|\!)$ to denote such a pair:

$$(\!|x|\!) \triangleq \langle X, x - X\rangle$$

The difference between authenticated sharings (Section 4.1) and additive sharings is that $\mathcal{P}$ does not know semantic values corresponding to additive sharings.

The parties can operate over additive sharings in the same way they can authenticated sharings: namely, we induce a vector space structure over additive sharings such that parties can add, subtract, multiply by public constants, and construct sharings of constants. Additionally, the parties can operate non-linearly by vector-scalar multiplication where $\mathcal{P}$ chooses the scalar. The needed protocol is *identical* to the vector-scalar protocol reviewed in Section 4.1.

Finally, $\mathcal{V}$ can construct a sharing $(\!|x|\!)$ for a value $x \in \mathbb{Z}_p$ that he chooses. To do so, $\mathcal{V}$ simply samples a uniform mask $M \in_\$ \mathbb{Z}_p$ and sends to $\mathcal{P}$ $x - M$.

4.5 Additive sharing permutations programmed by $\mathcal{P}$

In our construction, $\mathcal{V}$ chooses random masks that are used to authenticate the RAM content. $\mathcal{P}$ is then given the opportunity to arrange these masks as she likes so that she can implement the RAM access order. So, we need a capability by which $\mathcal{P}$ can rearrange $\mathcal{V}$'s chosen masks. The parties thus construct additive shares of the masks which can then be manipulated by $\mathcal{P}$.

More precisely, $\mathcal{V}$ chooses an array of random masks $K \in_{\$} (\mathbb{Z}_p^s)^n$, and the random masks are shared such that the parties hold $⦇K⦈$. Now, the parties must compute $⦇\pi(K)⦈$ for π chosen by $\mathcal{P}$. To apply an arbitrary permutation, we employ a particular circuit construction called a Waksman permutation network [Wak68]. This recursively constructed circuit builds a permutation of n elements from many permutations of two elements: i.e., from 'swap' gates. In our context, a swap gate allows $\mathcal{P}$ to conditionally swap two shares $⦇a⦈$ and $⦇b⦈$ based on her private bit $r \in \{0, 1\}$. Precisely, the gate is specified as follows:

$$swap(r, a, b) \triangleq \begin{cases} (a, b) & \text{if r = 0} \\ (b, a) & \text{otherwise} \end{cases}$$

To implement this gate, the parties compute a conditional difference $⦇\delta⦈ \triangleq r⦇a - b⦈$ and output the pair $⦇a - \delta, b + \delta⦈$. A swap gate is computed by a single vector-scalar multiplication and linear operations. The gate can be computed even though $\mathcal{P}$ knows neither a nor b.

A permutation network on n elements (where n is a power of two) consumes $n \log n - n + 1$ swap gates; hence we use $n \log n - n + 1$ oblivious transfers.

5 Technical Overview

In this section, we give high level intuition sufficient to understand our approach.

Informally, there are three types of attacks on ORAM $\mathcal{P}$ may attempt in her ZK proof: 1) modify a memory value by forging an authentication code, 2) return a stale value, or 3) return a valid authenticated value from a wrong location. The last attack is easily prevented by storing each array index as an authenticated value alongside the corresponding RAM element, and checking it on each access, a standard technique used, e.g., in [HK20a]. In this overview and in the formal constructions we focus on addressing issue 1) value modification. Preventing the return of stale values is achieved by enforcing a key invariant that a valid authenticated element cannot be stored in more than one place; we point this aspect out as we discuss how to ensure value integrity.

As a thought experiment, suppose that $\mathcal{V}$ and $\mathcal{P}$ both know the array access order; we will soon remove this restriction. That is, they know *a priori* the locations of *each* array read and write. Further, suppose that each array element is stored as an authenticated secret share (Section 4.1) held by both parties. That is, for an array A, its value at each index i is formatted as follows:

$$⟦A[i]⟧ = \langle K[i], A[i]\Delta - K[i] \rangle,$$

where $K[i]$ is a uniform mask chosen by $\mathcal{V}$. Suppose on the jth array access, the parties wish to access array slot i. This case is easy: each player can simply read from RAM slot i in their local memory, and use the already-authenticated array element as needed in the proof.

Of course, we want to access RAM in an order unknown to $\mathcal{V}$. Here we run into a problem: on an access of position i, $\mathcal{P}$ can still read $A[i]\Delta - K[i]$ from her local array, but $\mathcal{V}$ does not have sufficient information to align the matching mask $K[i]$. Further, $\mathcal{V}$ cannot be allowed to learn the accessed position i, since this would give her information about the access order.

Instead of giving $K[i]$ to $\mathcal{V}$, we instead allow $\mathcal{V}$ to use a fresh mask $M[j]$ and convey the appropriate matching mask to $\mathcal{P}$. Specifically, we arrange that $\mathcal{P}$ will obtain $K[i] - M[j]$. Given this information, the parties compute:

$$\langle M[j], (A[i]\Delta - K[i]) + (K[i] - M[j])\rangle = \langle M[j], A[i]\Delta - M[j]\rangle = [\![A[i]]\!]$$

This authenticated secret share can be used as a wire in the ZK circuit.

The remaining task is to show *how* these mask differences are securely conveyed to $\mathcal{P}$. We present our solution in several steps. First, we present solutions that allow for RAMs with constrained access orders; these initial constructions do not allow arbitrary RAM reads/writes. Then, we use these constrained constructions as building blocks upon which we achieve general purpose ORAM.

Read-once RAM. As a simplifying assumption, consider an n-element RAM that is preloaded with authenticated shares. Further, suppose the program will read each RAM slot exactly once, though the order in which these reads occur is unconstrained and is known to $\mathcal{P}$. In this case, the RAM's read order can be described by a *permutation* π on n elements that maps the time of each access to the accessed array index.

If we consider all n reads simultaneously, then our problem becomes one of delivering to $\mathcal{P}$ a sequence of n mask differences $K[i] - M[j]$, while hiding the access order from $\mathcal{V}$. To do so, $\mathcal{V}$ distributes to the two parties *additive secret shares* of the elements of the array of masks K: the parties hold $(\!|K|\!)$. Let π specify the permutation on A defining the RAM access order. The parties securely compute $(\!|\pi(K)|\!)$ using the permutation protocol described in Section 4.5. Informally, this permutation aligns the elements of K, which were originally in array order, with the order of accesses.

If we recall the syntax of an additive share $(\!|\pi(K)[j]|\!)$, we find that $\mathcal{P}$'s share has nearly the form that we need:

$$(\!|\pi(K)[j]|\!) = \langle Q[j], \pi(K)[j] - Q[j]\rangle = \langle Q[j], K[i] - Q[j]\rangle,$$

where $Q[j]$ is a uniform mask.

So far, the access masks M are *unconstrained*. Thus, $\mathcal{V}$ simply sets $M[j] = Q[j]$, and now each of $\mathcal{P}$'s share of the permuted array has *exactly* the form needed to align her share with that of $\mathcal{V}$. This implements read-once RAM: the parties can read an array of n elements in any order specified by $\mathcal{P}$.

swordRAM. Read-once RAM assumes that the array is preloaded with values. We also need a capability to write new RAM elements. Thus, we extend the above read-once RAM to allow for writes. However, the write capability we add is *highly constrained*: the parties must agree on and both know the order in which the array contents are written. For concreteness, we use a *sequential* write order, meaning that the jth write stores an element in the jth array slot. Array reads and writes may be arbitrarily interspersed with the restriction that each read occurs after the write to the accessed slot. As with our read-once RAM, we enforce that the program must read each array slot exactly once. We call this intermediate RAM a swordRAM (Sequential-Write, One-time ReaD RAM).

With the idea for read-once RAMs established, swordRAM is trivial. As argued in the beginning of this section, if each party knows the RAM access order, our task is easy: the parties trivially obtain matching authentication codes. Thus, swordRAM writes are simple, since both parties agree that the elements should be written sequentially, and hence the order is known to each. There is one subtlety in aligning the authentication masks used in RAM writes with the array slot masks $K[i]$, but this is easily addressed. Specifically, $\mathcal{V}$ simply sends the difference between the two masks to $\mathcal{P}$ on each RAM write.

General Purpose ZK ORAM. swordRAMs are highly restrictive. Nevertheless, there is an efficient reduction from general purpose RAM to swordRAM. We call this reduction PrORAM. A PrORAM of n elements is built on a swordRAM of $2n$ elements. There is no single one-to-one mapping from PrORAM slots to swordRAM slots. Rather, the swordRAM should be viewed as a *running log* of the PrORAM accesses; each PrORAM access corresponds to a single write and a single read in the swordRAM. At all times, we ensure that there are exactly n swordRAM slots that have been written to but not yet read, and it is exactly these n slots that hold the current PrORAM content. To track the relationship between PrORAM slots and swordRAM slots, the prover $\mathcal{P}$ maintains a clear-text data structure that we refer to as the *timetable*. A timetable $\mathcal{T}$ maps each PrORAM index i to the swordRAM slot where that element is currently stored.

The PrORAM is maintained as follows:

- To **initialize** a size-n PrORAM we perform a sequence of n writes to a fresh capacity-$2n$ swordRAM. Correspondingly, $\mathcal{P}$ initializes $\mathcal{T}$: at initialization, each PrORAM slot i is stored in swordRAM slot i.
- To **access** RAM slot i, $\mathcal{P}$ first looks up $\mathcal{T}[i]$ and reads from the corresponding swordRAM slot. Because of swordRAM's tight restrictions, this read 'exhausts' the accessed swordRAM slot, and so the parties must write back an element to the array. In the case of RAM write, the write-back element will be the written element. In the case of a RAM read, the write-back element will be the same element that was read. $\mathcal{P}$ then updates $\mathcal{T}$, indicating that PrORAM slot i is now stored in the newly written swordRAM slot.
- Because the number of reads/writes to a swordRAM are bounded, we must periodically **refresh** the PrORAM. Each PrORAM access consumes one swordRAM read and one swordRAM write. After n PrORAM accesses, we exhaust all $2n$ available swordRAM writes (recall, n writes were used to initialize) and n of the available $2n$ swordRAM reads. The remaining n reads suffice for us to fetch the current PrORAM content and store it into a freshly initialized swordRAM. By doing so, we "refresh" the PrORAM and are ready for n more accesses.

The **crucial point** is that because $\mathcal{P}$ knows the entire PrORAM access order $\mathcal{O}$ in advance, she can play out the above reduction "in her head" to obtain the

- Inputs: Parties agree on a swordRAM capacity n and a slot width s. $\mathcal{P}$ inputs a permutation on n elements π, denoting the order in which she wishes to read swordRAM elements.
- Outputs: Let $K \in_\$ (\mathbb{Z}_p^s)^n$ be uniform masks drawn by $\mathcal{V}$. Parties output a swordRAM $([\cdot], \pi, 0, K, [\cdot], (\!|\pi(K)|\!))$.
- Protocol:
  - $\mathcal{V}$ samples a length-n array of uniform values $K \in_\$ (\mathbb{Z}_p^s)^n$.
  - $\mathcal{V}$ constructs an additive sharing $(\!|K|\!)$ by sampling uniform masks $R \in_\$ (\mathbb{Z}_p^s)^n$ and sending $K - R$ to $\mathcal{P}$.
  - $\mathcal{V}$ and $\mathcal{P}$ compute $(\!|\pi(K)|\!)$ via a permutation network (see Section 4.5).
  - The swordRAM $([\cdot], \pi, 0, K, [\cdot], (\!|\pi(K)|\!))$ is now defined; the parties output their respective components.

Fig. 1. Initializing an empty capacity-n swordRAM. The parties output a swordRAM that encodes an empty array and that is ready for n writes and n reads. The n reads will happen as specified by the the access order π.

corresponding read order π for the underlying swordRAM. π is then used to initialize a swordRAM that will precisely service the access order $\mathcal{O}$.

Efficiency. PrORAM is efficient. Essentially the only cost is in permuting additive shares of the array K. For every n PrORAM accesses we initialize $2n$ swordRAM reads and thus consume a permutation of $2n$ masks. A permutation of $2n$ elements costs $2n \log 2n - 2n + 1$ OTs via a permutation network, and hence each PrORAM access consumes amortized $2 \log n$ OTs.

The remainder of this paper presents the above in technical detail.

6 PrORAM Formal Constructions

In this section, we present PrORAM in formal detail. Section 7 formalizes our construction's security.

6.1 swordRAM

Recall from Section 5 that we decompose the problem of building a RAM into two parts: first we construct a 'sequential write, one-time read RAM' (swordRAM) that only supports one read and one write per RAM slot, and where writes must occur in sequential order. Then we build a general purpose ORAM on top of swordRAM. We therefore start by defining swordRAM. Syntactically, a capacity-n swordRAM is a six-tuple:

$$(A, \pi, r, K, [\![A]\!]_K, (\!|\pi(K)|\!))$$

Each of these elements are as follows:

- $A \in (\mathbb{Z}_p^s)^*$ denotes the cleartext array encoded by the swordRAM. As we write to the swordRAM, A will grow in length. A is known only to $\mathcal{P}$.

- Inputs: Parties input a capacity-n swordRAM:

$$(A, \pi, r, K, \llbracket A \rrbracket_{K]}, (\!|\pi(K)|\!))$$

Let i be the read index: $i \triangleq \pi[r]$. It is illegal to call this functionality if $i \geq w$.
- Outputs: Parties output the read value $\llbracket A[i] \rrbracket$ and the updated swordRAM:

$$(A, \pi, r+1, K, \llbracket A \rrbracket_{K]}, (\!|\pi(K)|\!))$$

- Protocol:
  - Consider the sharing $(\!|\pi(K)[r]|\!) = (\!|K[i]|\!)$. Suppose $(\!|K[i]|\!) = \langle M, K[i] - M \rangle$. Note, $\mathcal{V}$ knows M: he simply looks up the rth element of his share of $(\!|\pi(K)|\!)$.
  - $\mathcal{P}$ fetches her share $\llbracket A[i] \rrbracket_K = A[i]\Delta - K[i]$.
  - Parties compute and output:

$$\langle M, (A[i]\Delta - K[i]) + (K[i] - M) \rangle = \langle M, A[i]\Delta - M \rangle = \llbracket A[i] \rrbracket$$

  - Parties increment r and output the updated swordRAM.

Fig. 2. Reading from a swordRAM. This procedure does not take an index as an argument. Rather, the index is defined by the permutation π chosen at initialization (cf. Figure 1).

- Inputs: Parties input a capacity-n swordRAM:

$$(A, \pi, r, K, \llbracket A \rrbracket_{K]}, (\!|\pi(K)|\!))$$

It is illegal to call this functionality if $|A| \geq n$. The parties input a sharing of a value a with mask M: $\llbracket a \rrbracket_M$.
- Outputs: Parties output the updated swordRAM:

$$(A \mid [a], \pi, r, K, \llbracket A \mid [a] \rrbracket_K, (\!|\pi(K)|\!))$$

I.e., the parties output a swordRAM where a is appended to A.
- Protocol:
  - Let $w = |A|$. Recall, the mask for swordRAM slot w is $K[w]$ known to $\mathcal{V}$. $\mathcal{V}$ sends to $\mathcal{P}$ the mask difference $M - K[w]$.
  - $\mathcal{P}$ computes:

$$(a\Delta - M) + (M - K[w]) = a\Delta - K[w]$$

    This is $\mathcal{P}$'s share of $\llbracket a \rrbracket_{K[w]}$. $\mathcal{P}$ appends a to A and appends her share of $\llbracket a \rrbracket_{K[w]}$ to the encrypted array.
  - Both parties output the updated swordRAM.

Fig. 3. Writing to a swordRAM. Recall that writes to swordRAM are *sequential*: the shared element a is appended to the array A.

- π is a permutation on n elements. π denotes the *read order* of the swordRAM. π is known only to $\mathcal{P}$. Note, the read order does not fully specify the *access* order, as writes may be arbitrarily interspersed with the constraint that each element is written before it is read.

- $r \in \mathbb{N}$ denotes the number of swordRAM reads that have occurred so far. In a valid swordRAM, $r \leq |A| \leq n$. Both $\mathcal{P}$ and $\mathcal{V}$ maintain local copies of r.
- $K \in_{\$} (\mathbb{Z}_p^s)^n$ is an n-element array with slots of size s, i.e. each slot K holds s values. $K[i]$ stores uniform masks used as swordRAM authenticators. $K[i]$ is drawn uniformly by $\mathcal{V}$ and is unknown to $\mathcal{P}$. We need more than one mask per swordRAM slot to support arrays of more general objects. In particular, in our RAMs we operate with value-index tuples (v, i), which allows us to perform an index check, preventing $\mathcal{P}$ from providing an invalid permutation and illegally substituting one RAM value for another.

  Although we use s masks for a single RAM slot, we are careful that any operations the parties perform are applied to the masks as a unit; hence, there is no opportunity for a cheating $\mathcal{P}$ to 'break apart' the contents within a single RAM slot.
- $[\![A]\!]_K$ is the authenticated secret sharing of A masked by K. Informally, this is the authenticated array. On a read, $\mathcal{P}$ indexes directly into this array and then aligns her share with $\mathcal{V}$'s (as described in Section 5).
- $(\!|\pi(K)|\!)$ is an additive secret sharing of the array K permuted according to π. These sharings are the values that $\mathcal{P}$ needs to align her shares with $\mathcal{V}$'s (as described in Section 5).

With syntax established, we describe operations over swordRAMs.

Initialize. Figure 1 lists the procedure for constructing a fresh swordRAM. At initialization, the encoded array A is empty (i.e., has size 0), so most of the swordRAM components are trivially initialized. The objective of initialization is to prepare for all n future reads. To do so, $\mathcal{P}$ provides as input the read order permutation π and $\mathcal{V}$ chooses a mask array K. The parties compute $(\!|\pi(K)|\!)$ via a permutation network (Section 4.5). This permutation provides to $\mathcal{P}$ the specific values that she needs to align her shares with $\mathcal{V}$'s on each read. We emphasize that swordRAM permutations account for almost all of our ORAM's cost.

Read. swordRAM reads (Figure 2) are entirely local operations: indeed, initialization already properly arranged that $\mathcal{P}$ will receive the correct mask alignment values on each read. $\mathcal{P}$ directly accesses the correct index of $[\![A]\!]_K$ and then aligns her share with $\mathcal{V}$'s using $(\!|\pi(K)|\!)[r]$.

Write. swordRAM writes (Figure 3) append values to the array A. The swordRAM authenticated array should be masked by the specific array K, but the parties write an arbitrary share $[\![a]\!]$. To properly store this value, $\mathcal{V}$ sends a difference between the mask on $[\![a]\!]$ and the target mask in K. $\mathcal{P}$ uses this value to align her share such that it can be properly appended.

As an aside, swordRAM performs no checking on the order in which $\mathcal{P}$ decides to read values: $\mathcal{P}$ freely chooses the read-order π. However, we next will perform a reduction from general purpose RAM to swordRAM. In this reduction, we explicitly include copies of each index identifier in the swordRAM. By this mechanism, the reduction fully constrains the permutation π, since the parties will check that each read yields the expected index identifier.

It will be convenient to abstract over some of the swordRAM detail. We give a shorthand for a swordRAM that encodes an array A with r remaining reads given by a read order π. Specifically we write $\rho(A, \pi, r)$:

$$\rho(A, \pi, r) \triangleq (A, \pi, r, K, [\![A]\!]_K, (\!|\pi(K)|\!))$$

where $K \in_\$ (\mathbb{Z}_p^s)^n$ is uniform and the masks on $(\!|\pi(K)|\!)$ are uniform.

6.2 swordRAM to PrORAM

Recall that we implement general purpose RAM by a reduction to swordRAM. We call this reduction PrORAM.

At a high level, a PrORAM implementing a size-n array operates in blocks of n accesses. Each block is handled by a distinct data structure, which is updated on each of the n accesses. After n accesses, we create a fresh data structure to support the next n accesses. We initialize the new structure by moving the contents of the old one, and then we retire the old data structure, and so on.

Each data structure is a capacity-$2n$ swordRAM (with accompanying metadata), which is initialized to contain the (current state of the) array A *in the canonical order* $A[0], ..., A[n-1]$. Of course, to initialize a swordRAM, we need an appropriate read order π. This permutation π must achieve two tasks: (1) it must encode the order of the next n accesses and (2) it must encode the order of the n reads needed to copy its content into the next swordRAM block *in canonical order* before being retired. That is, the first n (of the $2n$ total) reads of the capacity-$2n$ swordRAM service the n PrORAM requests for data, and the next n accesses read the array A as part of moving to the next PrORAM data structure. In total, there are $2n$ swordRAM reads, which can be encoded in a permutation π over $2n$ elements. We formally describe how to construct π based on the array's access order in Section 6.3.

PrORAM Syntax. We denote a PrORAM that encodes a cleartext array A with access order $\mathcal{O}$ by writing $\boxed{\text{A}, \mathcal{O}}$. A size-$n$ PrORAM is a four-tuple:

$$\boxed{\text{A}, \mathcal{O}} \triangleq (A, \mathcal{O}, \rho(H, \pi, r), \mathcal{T})$$

These elements are as follows:

- $A \in (\mathbb{Z}_p^s)^n$ is the cleartext content of the PrORAM. A is known only to $\mathcal{P}$.
- $\mathcal{O}$ is a list of all indexes accessed by the RAM and is known as the *access order*. $\mathcal{O}$ is maintained in cleartext by $\mathcal{P}$ and is unknown to $\mathcal{V}$. $\mathcal{P}$ can precompute $\mathcal{O}$ by running the proof in cleartext and logging all RAM accesses. For simplicity, assume $\mathcal{O}$ initially has length that is a multiple of n. $\mathcal{P}$ can pad $\mathcal{O}$ with extra zeros to reach the next multiple of n.
  As we perform accesses, the access order shrinks: each access removes the first element of $\mathcal{O}$ to reflect that the access has already been handled.
- $\rho(H, \pi, r)$ is a capacity-$2n$ swordRAM over an array H that we refer to as the *log*. Informally, the swordRAM logs each PrORAM access. The swordRAM's remaining reads $\pi[r..]$ correspond to $\mathcal{O}$. $\rho(H, \pi, r)$ is the authenticated component of PrORAM, and PrORAM's array accesses are ultimately authenticated via the mechanisms of this swordRAM.

```
schedule(𝒪) :
  ▷ Initialize a timetable to track which element will live where.
  ▷ Initially, the swordRAM will store elements in canonical order.
  𝒯 ← [0..n]
  return schedule − suffix(𝒪, 𝒯, n)

schedule − suffix(𝒪, 𝒯, r) :
  π ← 0^{r+n}     ▷ Initialize an array π to hold the remaining swordRAM reads
  ▷ Schedule a swordRAM read corresponding to each tth PrORAM access.
  for t ∈ [0..r] :
    i ← 𝒪[t]     ▷ Look up the target index of the tth access.
    slot ← 𝒯[i]     ▷ Look up the swordRAM slot that holds i.
    π[t] ← slot     ▷ slot should be read on the tth swordRAM read.
    ▷ Index i will be written back into the end of the swordRAM.
    ▷ Keep track of this write in the timetable.
    𝒯[i] ← 2n − r + t
  ▷ After all n accesses, we prepare to move elements to a fresh swordRAM.
  ▷ Thus, we schedule a read of each element in canonical order.
  for i ∈ [0..n] :
    slot ← 𝒯[i]     ▷ Look up the swordRAM slot that holds i.
    π[i + r] ← slot     ▷ slot should be read on the (i + n)th swordRAM read.
  return π
```

Fig. 4. Scheduling swordRAM accesses. schedule takes as an argument a PrORAM access order $\mathcal{O}$ and outputs a corresponding swordRAM read order permutation π. PrORAM supports schedules of arbitrary length, but schedule only sets up the next n accesses in the schedule, and hence only looks at the first n entries of $\mathcal{O}$.

schedule delegates to a more general procedure schedule − suffix which generates a length $r + n$ *suffix* of a read order permutation. While this more general call is never exercised in our execution (except directly via schedule), we use it to define validity of a general PrORAM state, in which some accesses may have occurred: a valid PrORAM must have a schedule equal to one (correctly) generated by schedule − suffix.

- $\mathcal{T}$ is the *timetable* maintained in cleartext by $\mathcal{P}$. The timetable maps each array index to the last timestep when that index was accessed. That is, for each array index i, $\mathcal{T}[i]$ is a pointer into the log denoting where $A[i]$ was last logged. The timetable is unknown to $\mathcal{V}$.

6.3 Scheduling the underlying swordRAM

Recall, we are working with an n-element PrORAM that facilitates operations on an n-element array A. In this section, we formally describe how to derive a swordRAM read order π given a length-n PrORAM access order. Recall from Section 6.2 that the permutation π must account both for the block of the next n PrORAM accesses and for the reads needed to copy array contents to a fresh PrORAM such that we can support more accesses.

Figure 4 presents schedule, an algorithm that computes π, the order in which the underlying swordRAM will obliviously read the elements of the log. schedule takes as input the given access order $\mathcal{O}$. swordRAM writes are sequential, and need not be scheduled, though the read schedule does depend on writes.

As explained in Section 6.2, each PrORAM data structure $\boxed{A, \mathcal{O}}$ is initialized with the array A in canonical order (initialization is discussed in Section 6.5).

To explain schedule, we first discuss how a single PrORAM access is mapped to the swordRAM. At initialization, the underlying capacity-$2n$ swordRAM stores all n elements of A in its first n available slots; the remaining n slots are not yet written and no reads have yet been used. Suppose that $\mathcal{P}$ wishes to read PrORAM slot $A[i]$. The swordRAM's read order permutation π should reflect this access: the first entry of π should indicate that slot i is read at time 0 (i.e., $\pi[0] = i$). Recall that swordRAM slots can be read only once. Therefore, to allow the PrORAM slot $A[i]$ to be read a second time, we must write back a value to the swordRAM. Because swordRAM writes occur sequentially, this write will place the new value into slot n. To account for this write, we should keep track of the new location of $A[i]$ which is done using a timetable $\mathcal{T}$. As a side remark, $\mathcal{T}$ is initialized to $[0, 1, ..., n-1]$, reflecting the fact that initially each element of A is stored in the swordRAM in canonical order.

Scheduling many accesses simply repeatedly applies the following basic procedure for accesses $j = 0, 1, \ldots, n-1$: Let i be the queried index on access j. We (1) look up the location of element i in the swordRAM based on $\mathcal{T}$, (2) update π such that slot i is read at time j (i.e., $\pi[j] = i$), (3) allocate the next available swordRAM write slot as the fresh location for element i, (4) update $\mathcal{T}$ to record that element i is stored in the fresh location.

schedule (Figure 4) implements this procedure. schedule accepts an access order $\mathcal{O}$ and outputs a permutation on $2n$ elements (encoded as an array) suitable for a swordRAM.

After allocating reads for the n accesses, schedule indicates that the last n entries in the permutation should match the current timetable. This detail is used to move the contents of an old data structure into a new one: after n accesses, we read the array contents in canonical order. The order of these last n reads is exactly what is stored in the final state of $\mathcal{T}$.

schedule highlights the key points of the reduction from RAM to swordRAM: map each array access to a swordRAM slot and continually update which array element is where. Of course, the reader must keep in mind the duality of our presentation as an iterative processing in response to queries, and the precomputed non-interactive one-shot schedule chosen before each block of n accesses.

6.4 PrORAM Validity

Before we specify PrORAM operations, we establish a validity condition that connects the PrORAM to its underlying swordRAM. This condition is the invariant that allows us to prove PrORAM is correct over many accesses.

As explained in Section 5, the swordRAM should be viewed as a log of the accesses to the PrORAM. PrORAM **validity** ensures that its swordRAM both (1) stores a log that properly reflects the PrORAM's current content and (2) has a read order that reflects PrORAM's future accesses.

Definition 1 (PrORAM Validity). *Let* $\boxed{A,\, \mathcal{O}} = (A, \mathcal{O}, \rho(H\,\pi, r), \mathcal{T})$ *be a size-n PrORAM. We say that this PrORAM is* ***valid*** *if:*

1. *For each PrORAM index* i*:*
$$H[\mathcal{T}[i]] = (A[i], i)$$
2. *Let* $w \triangleq |H|$ *be the number of elements written to the underlying swordRAM:*
$$\mathsf{schedule} - \mathsf{suffix}(\mathcal{O}, \mathcal{T}, n - w) = \pi[r..]$$

Less formally, these two conditions ensure the following:

1. If we look up each element's location in the timetable and then find each location in the log, then we recover the array A. This ensures that the swordRAM properly stores the array A. Note, we store each element $A[i]$ in a pair with its index i. This allows RAM accesses to check that the queried index matches the stored index, ensuring that $\mathcal{P}$ cannot substitute one RAM element for another.
2. If we construct a partial swordRAM schedule from the access order and the current timetable, then we obtain a new copy of the remaining swordRAM read order. This ensures that the remaining swordRAM reads properly reflect the array access order $\mathcal{O}$.

6.5 PrORAM Operations

Figures 5 to 7 list the operations over PrORAMs:

- Figure 5 indicates how a new PrORAM is initialized. The parties select an array of n sharings $[\![A]\!]$ as the initial array state, then sequentially write these elements into a fresh swordRAM. The procedure also sets up the swordRAM schedule and $\mathcal{P}$'s timetable $\mathcal{T}$. The swordRAM schedule is set using $\mathsf{schedule}$, and at initialization each PrORAM slot lives in the corresponding swordRAM slot: $\mathcal{T}$ is initialized to $[0, 1, ..., n-1]$.
- Figure 6 indicates how the parties access a PrORAM index. To access element i, the parties first read from the underlying swordRAM and retrieve a pair $[\![A[i], i']\!]$. The parties check that $i = i'$ by opening $\mathcal{P}$'s share of $i - i'$. This check ensures that $\mathcal{P}$ cannot substitute one array value for another.

- INPUTS: Parties input an array of n authenticated secret shares $[\![A]\!]$ that form the initial state of the array (for example, parties might use $[\![0]\!]^n$). $\mathcal{P}$ inputs the array access order $\mathcal{O}$.
- OUTPUTS: Parties output a valid initialized PrORAM:

$$(A, \mathcal{O}, \rho(H, \pi, 0), \mathcal{T})$$

  where $\mathcal{T} = [0, 1, ..., n-1]$ and the log H is equal to A.
- PROTOCOL:
  - $\mathcal{P}$ initializes her timetable $\mathcal{T}$ as $[0, 1, ..., n-1]$. That is, in the initial state of the PrORAM, each index i is in log slot i.
  - $\mathcal{P}$ computes $\pi \triangleq \mathsf{schedule}(\mathcal{O})$. π is the swordRAM read order.
  - Parties initialize an empty swordRAM to hold the log H; $\mathcal{P}$ uses π to perform this initialization.
  - Parties perform n writes to the swordRAM where the ith write stores the pair $[\![A[i], i]\!]$ (each i is a public constant, so the parties use the protocol's support for constants to encode these indexes). After all n writes, the swordRAM holds A in order, where each slot is explicitly marked with its index.
  - The PrORAM $(A, \mathcal{O}, \rho(A, \pi, 0), \mathcal{T})$ is now defined; the parties output their respective components.

Fig. 5. The PrORAM initialization procedure $\mathsf{initialize}$. $\mathsf{initialize}$ takes as arguments (1) an authenticated size-n array $[\![A]\!]$ and (2) an access order $\mathcal{O}$. $\mathsf{initialize}$ outputs a fresh PrORAM $\boxed{\text{A, }\mathcal{O}}$.

- Figure 7 is a helper procedure that allows the parties to refresh the PrORAM after every n accesses. To perform this refresh, the parties read the latest copy of every RAM slot from the swordRAM, then write these values back into a fresh swordRAM. We call the refresh procedure once every n accesses.

Crucially, each PrORAM operation preserves validity. We argue this formally in our proof of correctness.

Implementing read and write. access takes a general function f as an argument; accessing $A[i]$ also writes back $f(A[i])$. We quickly show that this is sufficient to implement the standard read and write array operations:

$$\mathsf{read}(\boxed{\text{A, }\mathcal{O}}, [\![i]\!]) \triangleq \mathsf{access}(\boxed{\text{A, }\mathcal{O}}, [\![i]\!], [\![x]\!] \mapsto [\![x]\!])$$
$$\mathsf{write}(\boxed{\text{A, }\mathcal{O}}, [\![i]\!], [\![y]\!]) \triangleq \mathsf{access}(\boxed{\text{A, }\mathcal{O}}, [\![i]\!], [\![x]\!] \mapsto [\![y]\!])$$

To implement read, we call access with the identity function: read simply writes back the read element. To implement write, we call access with a constant function that ignores the read element and returns the written element y.

Taking an arbitrary function is flexible. For example, we can implement an increment function that in-place updates an array slot:

$$\mathsf{increment}(\boxed{\text{A, }\mathcal{O}}, [\![i]\!]) \triangleq \mathsf{access}(\boxed{\text{A, }\mathcal{O}}, [\![i]\!], [\![x]\!] \mapsto [\![x+1]\!])$$

Thus, we can mutate an array value without using two RAM accesses.

- Inputs: Parties input:
  1. A size-n PrORAM $\boxed{\text{A, }\mathcal{O}}$.
  2. A shared index $[\![i]\!]$.
  3. An agreed upon function f used to update the selected element. f should be described as a circuit computable by the algebraic protocol (Section 4.1).
- Outputs: Parties output:
  1. The selected value $[\![A[i]]\!]$.
  2. The updated array $\boxed{\text{A[i := f(A[i])], }\mathcal{O}\text{[1..]}}$.
- Protocol:
  - Let $\boxed{\text{A, }\mathcal{O}} = (A, \mathcal{O}, \rho(H, \pi, r), \mathcal{T})$.
  - If $|H| = 2n$, parties call refresh (Figure 7) and replace the PrORAM by the output of the refresh operation.
  - Parties perform a read on $\rho(H, \pi, r)$ (Figure 2). Let $\rho(H, \pi, r+1)$ be the updated swordRAM and let $[\![x, i']\!]$ be the read value.
  - $\mathcal{P}$ opens her share of $[\![i - i']\!] = [\![0]\!]$. If $\mathcal{P}$'s share is not a share of 0, $\mathcal{V}$ aborts. This check prevents $\mathcal{P}$ from accessing an element $A[j]$ for $j \neq i$.
  - Parties jointly compute $[\![f(x)]\!]$ via the algebraic protocol.
  - Parties write $[\![(i, f(x))]\!]$ to $\rho(H, \pi, r+1)$ (Figure 3). Let $\rho(H|f(x), \pi, r+1)$ be the updated swordRAM.
  - Parties output $[\![x]\!]$.
  - Parties output $(A[i := f(A[i])], \mathcal{O}[1..], \rho(H|f(x), \pi, r+1), \mathcal{T}[i := |H|])$. That is, they output $\boxed{\text{A[i := f(A[i])], }\mathcal{O}\text{[1..]}}$, which is the PrORAM updated to include the new write.

Fig. 6. PrORAM access procedure access. access performs the following functions: (1) it looks up and outputs the queried element $[\![A[i]]\!]$, (2) it computes $[\![f(A[i]]\!]$ for arbitrary circuit-encoded function f, and (3) it writes $[\![f(A[i]]\!]$ back to the array. If $\mathcal{O}[0] \neq i$ (that is, if $\mathcal{P}$ tries to use a bad read order), then $\mathcal{V}$ will abort.

- Inputs: Parties input a valid size-n PrORAM $(A, \mathcal{O}, \rho(H, \pi, n), \mathcal{T})$ such that $|H| = 2n$; i.e., the underlying swordRAM has no writes and n reads remaining.
- Outputs: Parties output a valid, refreshed PrORAM $(A, \mathcal{O}, \rho(H', \pi', 0), \mathcal{T}')$ such that $|H'| = n$; i.e., the new underlying swordRAM has n writes and $2n$ reads remaining.
- Protocol:
  - Parties perform n swordRAM reads on $\rho(H, \pi, n)$. Because of the validity condition (Definition 1) and the definition of schedule (Figure 4), these n reads fetch the array content: the parties hold $[\![A]\!]$.
  - Parties call the PrORAM initialize procedure (Figure 5) with $[\![A]\!]$ and $\mathcal{O}$ and return the resulting PrORAM.

Fig. 7. PrORAM refresh procedure. PrORAM is built on top of swordRAM which allows only a bounded number of reads/writes. To allow many PrORAM accesses, we periodically *refresh*. The refresh procedure simply reads the content of the old swordRAM into an array, then initializes a fresh PrORAM with the result.

6.6 PrORAM Formal Properties

In this section, we state PrORAM's formal properties. **Due to lack of space, we defer full proofs of these properties to Supplementary Material.**

$\mathsf{initialize}$ and access maintain validity:

Theorem 1 (Initialize Correctness). *Let $[\![A]\!]$ be an authenticated share of an array of n elements and let $\mathcal{O}$ be an arbitrary access order over n elements.*

$$\mathsf{initialize}([\![A]\!], \mathcal{O}) = \boxed{A,\, \mathcal{O}}$$

where $\boxed{A,\, \mathcal{O}}$ is a valid PrORAM.

Theorem 2 (Access Correctness). *Let $\boxed{A,\, \mathcal{O}}$ be a valid n-element PrORAM. Let $j \triangleq \mathcal{O}[0]$. Let $[\![i]\!]$ be a shared RAM index, and let f be a publicly agreed function. If $i = j$ (i.e., if the shared RAM index matches the access order), then the following holds:*

$$\mathsf{access}(\boxed{A,\, \mathcal{O}}, [\![i]\!], f) = ([\![A[i]]\!], \boxed{A[i := f(A[i])],\, \mathcal{O}[1..]}),$$

where $\boxed{A[i := f(A[i])],\, \mathcal{O}[1..]}$ is a valid PrORAM.

In short, we show that the operations update the timetable/schedule and appropriately make use of swordRAM such that validity is maintained.

PrORAM is also concretely efficient:

Theorem 3. *The procedure* access *(Figure 6) invoked on a size-n PrORAM consumes amortized $2 \log n$ oblivious transfers of length 2σ secrets. Additionally, each access transmits amortized 8σ bits.*

In short, we inspect the PrORAM algorithms for communication cost, then amortize costs across each block of n accesses.

7 A Complete ZKP System and Security Proofs

Our approach to proving security. Our PrORAM naturally integrates with ZKP systems based on authenticated shares, such as the ZKP system of [HK20a]. To define and prove security of a ZK ORAM construction, including our PrORAM, one needs to set up a general ZK proof environment which can generate arbitrary RAM query patterns. The ZKP system of [HK20a] provides a simple, general, and efficient environment. We embed PrORAM directly into this protocol, and we state and prove the security properties of the resulting system.

We list the following benefits from taking this route:

1. We construct a *complete* PrORAM-based ZKP system.
2. [HK20a], and hence our complete system, is concretely efficient.
3. As discussed next, we can reuse the clean and powerful garbled circuit-based ZK framework of [JKO13,FNO15].
4. We obtain a simple formalism that can be easily generalized/plugged in other systems (separate proofs are required, but often may be modeled on our proof blueprint).

7.1 Casting as a Garbling Scheme

Much like [HK20a], we cast our system as a Garbling Scheme (GS), and thus are able to reuse the convenient and powerful framework of [JKO13,FNO15]. Their framework plugs a custom GS (satisfying certain requirements) into their protocol; the instantiated constant round protocol achieves malicious-verifier ZK.

In the following, we derive notation from [BHR12], but include changes proposed by recent works that separate the circuit's logical description from *GC material* [HK20c,HK20b]. We explicitly include both the GC material M and the computed circuit $\mathcal{C}$ as arguments to our GS functions.

Before continuing, we discuss the correspondence of our system to a garbling scheme, as this correspondence may *a priori* be unintuitive; after all, we do not construct encryptions of logical gates which are the hallmark of garbled circuits. Nevertheless, our construction does have components that map cleanly to a GS:

Garbled input labels. In a GS, the GC evaluator receives *garbled input labels*. These labels are typically encryption keys that correspond to the logical values on the input wires. The collection of all input labels is called the *encoding* (denoted e), and in most protocols the parties run OTs to send a selection of input labels (a subset corresponding to the player's input) from the encoding to the evaluator. Our labels are more naturally understood as *authentication keys*, rather than encryption keys. We send particular authentication mask differences via OT to enable the authentic multiplication of shares (see Section 4.1). The collection of all OT messages used for multiplications forms our encoding e.

Garbled material. In a GS, the GC evaluator receives an extra string that does not depend on her input and is used to evaluate the GC. This string is called the *material* (denoted M), and is typically a collection of encrypted truth tables. While we do not encrypt truth tables, we do send fixed values from $\mathcal{V}$ to $\mathcal{P}$ to initialize additive shares and to execute writes to swordRAMs (see Figure 3). The collection of these extra messages is our material M.

Garbled output label. Similar to the input encoding e, GSs also require an output decoding (denoted d). In the [JKO13] framework, d is a single, unforgeable value that indicates a proof; $\mathcal{V}$ simply checks that $\mathcal{P}$ indeed constructed d to become convinced. In our construction, the string d is the hash digest of all of $\mathcal{P}$'s opened shares (see Section 4.1).

Achieving verifiability. The [JKO13] framework requires a GS to be verifiable. Informally, this provides for a way to "open" the garbled function to prove that it was constructed correctly. One natural way to achieve this, which we adopt, is for all of $\mathcal{V}$'s randomness be derived from a seed S. Revealing S allows $\mathcal{P}$ to verify the garbled function. GSs and the [JKO13] framework do not provide a side channel for $\mathcal{V}$ to deliver S to $\mathcal{P}$. Therefore, we use e for this purpose: we simply XOR secret share S and append the shares to the labels of wire 1 of the circuit. This way, S remains protected until it is opened by $\mathcal{V}$.

7.2 The [JKO13] ZK Framework

To plug a construction into [JKO13]'s ZK protocol, we must prove that the construction is a **verifiable garbling scheme**. A verifiable garbling scheme is a tuple of six algorithms (see [BHR12,JKO13] for precise syntax and formalization details):

$$(\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \mathsf{Ve})$$

The first five algorithms define a garbling scheme [BHR12], while the sixth adds verifiability [JKO13].

A garbling scheme specifies the functionality computed by $\mathcal{V}$ and $\mathcal{P}$. $\mathcal{V}$ uses Gb to construct material M, input encoding e, and output decoding d. Gb is computed by walking through the agreed proof circuit $\mathcal{C}$ gate-by-gate. In our construction, we simplify Gb by ensuring that all random values are chosen according to a single pseudorandom seed. Next, $\mathcal{V}$ uses OT to encode $\mathcal{P}$'s witness according to e. En specifies what these OTs should accomplish: it maps $\mathcal{P}$'s input space to a concrete choice of encoding, specifying the particular values in e that $\mathcal{P}$ should receive for each of her inputs. Upon receiving material M and an encoded witness, $\mathcal{P}$ uses Ev to authentically compute the circuit gate-by-gate. At the end of a ZK proof, $\mathcal{P}$ constructs a particular output value which is first committed and later sent to $\mathcal{V}$. $\mathcal{V}$ then calls De, which checks that the received value is exactly equal to the output decoding d; if not, $\mathcal{V}$ aborts.

The steps described so far do not protect $\mathcal{P}$ from a cheating $\mathcal{V}$, who might maliciously construct e and M in order to leak $\mathcal{P}$'s input. Therefore, before opening her commitment, $\mathcal{P}$ rebuilds M, e, and d according to $\mathcal{V}$'s seed (which is sent after the commitment). $\mathcal{P}$ uses these reconstructed values to check that the messages received from $\mathcal{V}$ were honestly constructed. If so, she opens her commitment; if not, she aborts. Ve describes how $\mathcal{P}$ should reconstruct M, e, and d and how she should check that $\mathcal{V}$ did not cheat.

Finally, ev provides a *specification* against which the correctness of the garbling scheme can be checked: ev describes the cleartext semantics of the circuits manipulated by the GS.

A verifiable garbling scheme must be **correct**, **sound**, and **verifiable** (definitions are in Section 7).

7.3 Our Garbling Scheme and Its Security

Our garbling scheme is the arithmetic garbling scheme of [HK20a] augmented with PrORAM. The arithmetic circuit may arbitrarily issue calls to PrORAM's $\mathsf{initialize}$ and access functionalities (Figures 5 and 6).

Construction 1 (Our Garbling Scheme)**.** *Our garbling scheme is the six tuple of algorithms:*

$$(\mathsf{ev}, \mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \mathsf{Ve})$$

described below. Circuits handled by the garbling scheme allow (1) publicly agreed constant wire values, (2) addition gates, (3) subtraction gates, (4) scalar gates (which multiply a value by a public constant), (5) vector-scalar multiplication gates (where the scalar is chosen by $\mathcal{P}$), (6) opening gates (which force $\mathcal{P}$ to prove a share represents a specific constant), (7) array initialization gates, and (8) array access gates. Circuits thus include two types of wires: (1) algebraic wires that hold values in $\mathbb{Z}_p$ and (2) array wires that hold arrays of values in $\mathbb{Z}_p$.

Our circuits do not include standard multiplication gates, but recall (from Section 4.2) that standard multiplication gates are easily implemented on top of vector-scalar multiplication gates and opening gates.

We describe each of our garbling scheme procedures:

ev evaluates the ZK relation in cleartext and implicitly specifies the cleartext semantics of each gate type. Our gate types have natural semantics, for example addition gates indeed add their inputs.

Gb processes the circuit gate-by-gate. As it goes, it generates random values, obtained from expansion of a pseudorandom seed S. The procedure generates the mask differences that are $\mathcal{V}$'s OT inputs (i.e. the encoding e). Additionally, Gb generates the material M: when $\mathcal{V}$ constructs additive sharings and on swordRAM writes, Gb appends the 'sent' component of the sharing to accumulated string of material. To handle opening gates, the algorithm also accumulates, as it goes, the hash of the expected opened shares (that $\mathcal{V}$ expects from $\mathcal{P}$). The final value of this hash is decoding secret d.

Gb processes arithmetic gates according to the [HK20a] protocol (see Section 4.1). Array access gates are processed with our ORAM construction (Figures 5 and 6). Each of these gates is handled by running $\mathcal{V}$'s procedure.

As an additional detail, Gb includes in e two XOR secret shares of the pseudorandom seed S. We discuss this in Section 7.1 under **achieving verifiability**.

En describes which mask differences (for vector-scalar multiplication gates) $\mathcal{P}$ should receive according to her input. Looking at the procedure for vector-scalar multiplication (Section 4.1), En is the trivial mapping that indicates $\mathcal{P}$ should receive the left OT secret if her input is zero and the right OT secret otherwise (cf. Equation (1) in Section 4.1).

Ev is complementary to Gb. Like Gb, it processes the circuit gate-by-gate. On vector-scalar multiplication gates, Ev consumes encoded input delivered by En. On the construction of additive sharings/swordRAM writes Ev consumes material in M. On opening gates, Ev accumulates a hash of opened shares.

Ev handles each gate by running $\mathcal{P}$'s procedures as described in Section 4.1 and Figures 5 and 6.

De is a simple comparison: if the expected output d is equal to the provided hash, then the procedure accepts; otherwise it rejects (and $\mathcal{V}$ aborts).

Ve is implemented in the same manner as Gb: it uses the pseudorandom seed (included in e, see Section 7.1) to replay the actions of Gb. As it goes, it checks that the generated encoding e, material M, and decoding d are equal to the given values. If all values are equal, Ve accepts; otherwise it rejects (and $\mathcal{P}$ aborts).

We next formalize that Construction 1 is **correct**, **sound**, and **verifiable**. **Due to a lack of space, we prove each theorem in Supplementary Material.** These theorems, combined with Theorem 2 from [JKO13] and theorems in Section 6 imply the following:

Theorem 4 (Main Theorem). *In the OT-hybrid model, assuming a collision-resistant hash function and statistical security parameter σ, Construction 1 is a (malicious-verifier) ZKP system with soundness $O(2^{-\sigma})$. Circuits in the resulting* *system may construct and access random-access arrays, and each access to an array of size* n *consumes amortized* $2 \log n$ *OTs of length* 2σ *secrets.*

Definition 2 (Correctness). *A garbling scheme is* ***correct*** *if for all circuits* $\mathcal{C}$ *and all inputs* i *such that* $\mathcal{C}(i) = 1$*:*

$$(e, M, d) = \mathsf{Gb}(1^\sigma, \mathcal{C}) \implies \mathsf{Ev}(\mathcal{C}, M, \mathsf{En}(e, i), i) = d$$

Correctness enforces that GS correctly implements the specification ev.

Theorem 5. *Construction 1 is* ***correct****.*

In short, correctness follows from the correctness of [HK20a]'s arithmetic protocol and from the correctness of PrORAM (Theorems 1 and 2).

Definition 3 (Soundness). *A garbling scheme is* ***sound*** *if for all circuits* $\mathcal{C}$*, all inputs* i *such that* $\mathcal{C}(i) = 0$*, and all probabilistic polynomial time adversaries* $\mathcal{A}$ *the following probability is negligible in* σ*:*

$$Pr(\mathcal{A}(\mathcal{C}, M, \mathsf{En}(e, i)) = d : (e, M, d) \leftarrow \mathsf{Gb}(1^\sigma, \mathcal{C}))$$

Soundness ensures that a cheating $\mathcal{P}$ cannot forge a convincing proof.

Theorem 6 (Soundness). *Assuming the existence of collision-resistant hash functions, Construction 1 is* ***sound****.*

In short, soundness follows from the authenticity of secret shares. $\mathcal{P}$ cannot forge RAM values because each is masked by a distinct value chosen by $\mathcal{V}$.

Definition 4 (Verifiability). *A garbling scheme is* ***verifiable*** *if for all circuits* $\mathcal{C}$*, all inputs* i *such that* $\mathcal{C}(i) = 1$*, and all probabilistic polynomial time adversaries* $\mathcal{A}$ *there exists an expected polynomial time algorithm* Ext *such that the following probability is negligible in* σ*:*

$$Pr\left(\mathsf{Ext}(\mathcal{C}, M, e) \neq \mathsf{Ev}(\mathcal{C}, M, \mathsf{En}(e, i)) : (e, M) \leftarrow \mathcal{A}(1^\sigma, \mathcal{C}), \mathsf{Ve}(\mathcal{C}, M, e) = 1\right)$$

At a high level, in the [JKO13] protocol, $\mathcal{P}$ receives and evaluates GC and commits to her proof message. Then she is given $\mathcal{V}$'s private randomness used to construct the GC. $\mathcal{P}$ uses this randomness to check messages sent by $\mathcal{V}$. Verifiability ensures that this check is reliable in the following sense: $\mathcal{V}$ will learn nothing from the opened proof message because $\mathcal{P}$'s proof message can be reconstructed in polytime by Ext without $\mathcal{P}$'s witness. Altogether, verifiability ensures that the ZK protocol is secure against a malicious verifier.

Our construction takes a natural approach and derives all of $\mathcal{V}$'s randomness from a seed S, and then reveal S as part of the verification procedure Ve. To syntactically fit the conveyance of S into the [JKO13] framework, we include S in e. See discussion accompanying the protocol specification Construction 1. Note, opening all of $\mathcal{V}$'s private randomness is a natural protocol design decision, but is not required by the definition of verifiability (Definition 4).

Theorem 7 (Verifiability). *Construction 1 is* ***verifiable****.*

In short, verifiability follows relatively trivially from the fact that $\mathcal{V}$ chooses all randomness starting from a pseudorandom seed. We remind the reader that due to a lack of space, we prove each theorem in Supplementary Material.

8 Instantiation

We implemented PrORAM in 1300 lines of `C++`. Our implementation uses the recent and efficient correlated Ferret OT technique [YWL+20]. Thus, our implementation inherits Ferret's cryptographic assumptions: (1) learning parity with noise (LPN), (2) a tweakable correlation-robust hash function, and (3) a random oracle (RO). We use statistical security parameter $\sigma = 40$ and accordingly instantiate our prime field with modulus $p = 2^{40} - 87$, the largest 40 bit prime.

In the following section, we discuss an experimental evaluation of our implementation. All experiments were performed on a MacBook Pro laptop with an Intel Dual-Core i5 3.1 GHz processor and 8GB of RAM. We ran our experiments on a simulated LAN network featuring 1Gbps of bandwidth and 2ms latency.

9 Evaluation

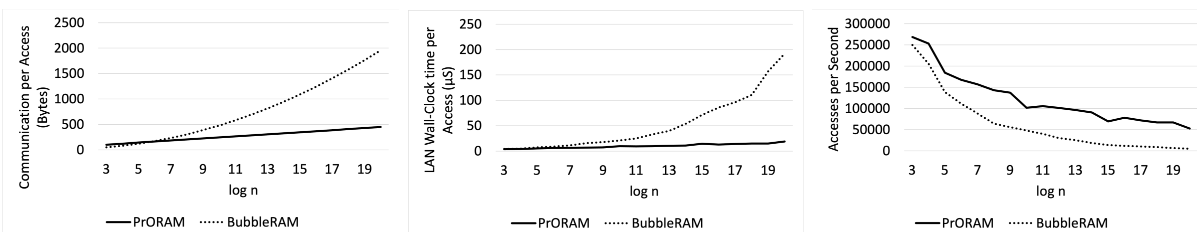

Fig. 8. Performance comparison of PrORAM against [HK20a]'s BubbleRAM. We plot performance as a function of the size of RAM n. Each experiment accessed the RAM 2^{20} times. We plot (1) the amortized communication cost of each access (left), (2) the amortized wall-clock time per access (center), and (3) the number of accesses performed per second (right). Center and right are different views of the same information.

In this section, we illustrate the performance of PrORAM by experimental evaluation. For comparison, we also ran BubbleRAM, a circuit-based ZK ORAM that was implemented as part of [HK20a]'s ZK construction. Since their construction is built on the same underlying arithmetic protocol, the comparison is direct. We emphasize that we implement both constructions in the same protocol and use the *same underlying OT protocol* (Ferret [YWL+20]); thus our experiments directly compare the ORAM techniques, not the environments they run in. Our comparison highlights the low asymptotic and concrete costs of PrORAM.

We implemented both PrORAM and BubbleRAM and used them to evaluate a circuit which accesses an array 2^{20} times on random indexes. Of course, a more realistic use case would use the RAM in the context of a more complex circuit, but our goal is only to measure performance. We varied the size of the RAM n between 2^3 slots and 2^{20} slots. Each RAM slot holds a single $\mathbb{Z}_p$ element;

recall that, internally, the PrORAM also reserves an extra slot to store the index identifier. Hence, internally the PrORAM slots have width two; BubbleRAM uses the same trick and hence also has slots of width two. We measured both the total communication transmitted between $\mathcal{P}$ and $\mathcal{V}$ and the wall-clock time needed to complete the entire proof. Figure 8 plots the results of these experiments.

Communication improvement. Our communication improvement follows naturally from our improved asymptotics: BubbleRAM incurs $1/2 \log^2 n$ OTs per access while we incur only $2 \log n$. In addition to the OTs, our $\mathcal{V}$ also sends an additional eight $\mathbb{Z}_p$ elements per RAM access: four to convey shares of K to $\mathcal{P}$ before permuting and four for the two swordRAM writes.

PrORAM outperforms BubbleRAM for $n > 2^5$. At $n = 2^{20}$, communication is improved by $4.36\times$.

Wall-clock time improvement Our wall-clock time improvement is far more dramatic than our communication improvement.

Both BubbleRAM and PrORAM primarily involve applying Waksman permutation networks to an array of shared values. However, PrORAM applies only a single permutation to prepare for n accesses. In contrast, BubbleRAM applies a permutation on *each* access (though the permutations vary in size). Waksman networks are not cache friendly. The network involves swapping (via algebra) data between disparate locations in the array of shares. Thus, computing the network causes many cache misses and is expensive. Because we dramatically reduce the number of permutations, we see a corresponding performance boost. At $n = 2^{20}$, we improve over BubbleRAM by $10.6\times$.

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.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Heidelberg, May 2020.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. `https://eprint.iacr.org/2018/046`.

[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+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 401–414. ACM, January 2013.

[BFH+20] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 2025–2038. ACM Press, November 2020.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE Computer Society Press, May 2015.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[FNO15] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 191–219. Springer, Heidelberg, April 2015.

[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.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43, 01 1996.

[HK20a] 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.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Heidelberg, August 2020.

[HK20c] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Heidelberg, May 2020.

[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.

[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.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[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.

[RS19] Michael A. Raskin and Mark Simkin. Perfectly secure oblivious RAM with sublinear bandwidth overhead. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 537–563. Springer, Heidelberg, December 2019.

[SvS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2020/925, 2020. `https://eprint.iacr.org/2020/925`.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1607–1626. ACM Press, November 2020.

Supplementary Material

10 Proofs

Due to lack of space, we deferred our proofs from the main part of our paper. Here, we restate and prove each of our theorems.

Theorem 1 (Initialize Correctness). *Let $[\![A]\!]$ be an authenticated share of an array of n elements and let $\mathcal{O}$ be an arbitrary access order over n elements.*

$$\mathsf{initialize}([\![A]\!], \mathcal{O}) = \boxed{A,\ \mathcal{O}}$$

where $\boxed{A,\ \mathcal{O}}$ is a valid PrORAM.

Proof. By the correctness of swordRAM operations.

initialize (Figure 5) simply sequentially writes the n elements in $[\![A]\!]$ to a fresh swordRAM. The initial timetable $\mathcal{T} = [0, 1, ..., n-1]$ thus correctly indicates the location of each index i in the swordRAM. Moreover, the schedule $\pi = \mathsf{schedule}(\mathcal{O}[0..n], \mathcal{T}, n)$ satisfies validity by construction.

initialize is correct. □

Theorem 2 (Access Correctness). *Let $\boxed{A,\ \mathcal{O}}$ be a valid n-element PrORAM. Let $j \triangleq \mathcal{O}[0]$. Let $[\![i]\!]$ be a shared RAM index, and let f be a publicly agreed function. If $i = j$ (i.e., if the shared RAM index matches the access order), then the following holds:*

$$\mathsf{access}(\boxed{A,\ \mathcal{O}}, [\![i]\!], f) = ([\![A[i]]\!], \boxed{A[i := f(A[i])],\ \mathcal{O}[1..]}),$$

where $\boxed{A[i := f(A[i])],\ \mathcal{O}[1..]}$ is a valid PrORAM.

Proof. By validity of the input $\boxed{A,\ \mathcal{O}}$ and correctness of swordRAM operations.

At a high level, we show that accessing a valid PrORAM will return (1) a correct sharing of the accessed slot and (2) a valid, updated PrORAM.

Access is correct if $|H| < 2n$. Suppose that the PrORAM's internal log H has size less than $2n$; we address the case $|H| = 2n$ at the end of the proof.

The definition of validity (Definition 1) combined with the definition of schedule (Figure 4) ensures that the next item to be read from the swordRAM corresponds to array index i. More concretely, validity ensures that:

- $\mathcal{T}[i]$ denotes the index of the log H where $A[i]$ is stored: $H[\mathcal{T}[i]] = A[i]$.
- The first item in $\mathsf{schedule-suffix}(\mathcal{O}, \mathcal{T}, t)$ is $\mathcal{T}[i]$.
- The underlying swordRAM's schedule π matches the access order $\mathcal{O}$.

Put together, these facts ensure that the next element to read from H is the accessed element: $\pi(H)[r] = A[i]$. Since swordRAM reads are correct, the PrORAM access will output a correct sharing $[\![A[i]]\!]$.

The updated PrORAM is a valid encoding of $A[i := f(A[i])]$ with access order $\mathcal{O}[1..]$. First, condition (2) of validity (Definition 1) follows from iterative structure of $\mathsf{schedule}-\mathsf{suffix}$: the schedule for the updated PrORAM automatically matches the schedule for its underlying swordRAM. Second, condition (1) of validity follows from the fact that we write $f(A[i])$ back to the swordRAM. In particular, after the swordRAM read but before the swordRAM write, $\mathcal{T}$ correctly indicates the location of every array index except for the accessed index i. $\mathsf{schedule}-\mathsf{suffix}$ sets the new location for index i as the end of the log, which matches the fact that we sequentially write back the element $f(A[i])$ to the end of the log. Thus, the updated PrORAM is valid.

Refresh is correct. Now, suppose that the input PrORAM has log H with size $2n$. In this case, we refresh the PrORAM before the access (Figure 7). Thus, we must show that refresh correctly outputs a new valid PrORAM $\boxed{\text{A}, \mathcal{O}}$, but whose log has size n instead of $2n$. Recall that the current schedule π was computed according to $\mathsf{schedule}$ (see Figure 5). $\mathsf{schedule}$ computes a swordRAM schedule of n accesses, and then schedules accesses to each index i in canonical order. Thus, by validity, the last n scheduled swordRAM reads look up the current PrORAM content. Our refresh operation uses this fact to copy the PrORAM content to a temporary array. Note, this intermediate array does not support random access, and is just a shared array of authenticated values $[\![A]\!]$. The parties simply call $\mathsf{initialize}$ with content $[\![A]\!]$ (Figure 5), and then perform the access on the resulting PrORAM.

PrORAM is correct. □

Theorem 3. *The procedure* access *(Figure 6) invoked on a size-n PrORAM consumes amortized* $2\log n$ *oblivious transfers of length* 2σ *secrets. Additionally, each access transmits amortized* 8σ *bits.*

Proof. By amortizing OTs and transmissions across n accesses.

First consider $\mathsf{initialize}$ (Figure 5). $\mathsf{initialize}$ is called once to build a fresh PrORAM and then is called after every n accesses when the underlying swordRAM is filled. $\mathsf{initialize}$ of a size n PrORAM invokes a permutation on a size-$2n$ array via swordRAM initialization (Figure 1). This array's slots each contain two additive shares: one share holds the array content while the other holds the explicit index identifier used to ensure $\mathcal{P}$ cannot substitute one array value for another. The permutation over these slots is achieved by a Waksman permutation network [Wak68]. A network on $2n$ elements consumes $2n\log 2n - 2n + 1$ swap gates, each of which can be instantiated by a single vector-scalar multiplication (Section 4.5). Thus, we consume $2n\log 2n - 2n + 1$ vector scalar multiplications where each vector is of length two. Each multiplication is instantiated by a corresponding OT. By amortizing these OTs across n accesses, each access consumes $2\log n + 1/n$ OTs of length 2σ secrets.

On initialization, $\mathcal{V}$ sends to $\mathcal{P}$ her initial shares of $(\!|K|\!)$. This array share is of size $4\sigma n$: K is an array of $2n$ slots where each slot hold two shares. Amortized, each access therefore incurs an extra 4σ bits for initialization.

On each swordRAM write, $\mathcal{V}$ sends to $\mathcal{P}$ a mask difference (Figure 3). This difference is proportional to the size of swordRAM slots, and hence has size 2σ. Every n PrORAM accesses corresponds to $2n$ swordRAM writes, so each access costs amortized 4σ extra bits for writing.

swordRAM initializations and writes feature the only communication in our construction, so we have accounted for all communication cost. $\square$

Theorem 4 (Main Theorem). *In the OT-hybrid model, assuming a collision-resistant hash function and statistical security parameter σ, Construction 1 is a (malicious-verifier) ZKP system with soundness $O(2^{-\sigma})$. Circuits in the resulting system may construct and access random-access arrays, and each access to an array of size n consumes amortized $2\log n$ OTs of length 2σ secrets.*

Proof. Follows immediately from the other theorems proved in this Supplementary Material. $\square$

Theorem 5. *Construction 1 is **correct**.*

Proof. By the correctness of [HK20a]'s algebraic protocol and the correctness of PrORAM (Theorem 2).

Correctness follows trivially from the correctness of these subcomponents. The only noteworthy detail is that PrORAM accesses take as input/give as output authenticated secret sharings in the same format as used by algebraic operations (see Figure 6). Hence, PrORAM values may be interpreted as circuit wire values.

Construction 1 is correct. $\square$

Theorem 6 (Soundness). *Assuming the existence of collision-resistant hash functions, Construction 1 is **sound**.*

Proof. By the security of [HK20a]'s arithmetic representation and an argument that the random authentication masks $K \in_\$ (\mathbb{Z}_p^s)^n$ prevent the forgery of values. We first prove the non-RAM portions of the arithmetic protocol sound. Then we focus on PrORAM.

Each circuit wire value x is represented by an authenticated secret share $[\![x]\!] = \langle X, x\Delta - X\rangle$ for fixed uniform $\Delta \in_\$ \mathbb{Z}_p^\times$ and for $X \in \mathbb{Z}_p$. Crucially, X and Δ are unknown to $\mathcal{A}$. Given a share $x\Delta - X$, it is infeasible for $\mathcal{A}$ to forge a valid share $y\Delta - X$ for $x \neq y$: $\mathcal{A}$ does not know Δ, does not know X, and the multiples of Δ are distributed evenly over $\mathbb{Z}_p^\times$. Therefore, forging a specific sharing $y\Delta - X$ requires $\mathcal{A}$ to simply guess Δ, which succeeds with probability $\frac{1}{p-1}$. We choose $p > 2^{\sigma-1}$, and so inauthentic values can only be forged with probability negligible in σ. Vector space operations (addition, subtraction, multiplication by public constant) are computed locally by homomorphism, and so these operations are trivially sound (that is, the resulting output share is unforgeable in the above sense). Vector scalar multiplication requires the parties to communicate via OT, and the received messages are in $\texttt{En}(e, i)$. The received OT messages are randomized and cannot be used to forge authentic values. The output of a vector-scalar multiplication is a newly randomized authenticated secret share, and so multiplication is also sound. Finally, the protocol accumulates a hash digest of all opened shares; forging this hash requires breaking the collision resistance of a hash function and is infeasible. Thus, Construction 1's arithmetic operations are sound.

We now focus on PrORAM. In terms of soundness, PrORAM is mostly a straightforward reduction to swordRAM: each PrORAM access primarily delegates to a swordRAM read and a swordRAM write (and possibly a swordRAM initialization via refresh). One extra detail is that each PrORAM access checks that the looked up index matches the queried index. This check ensures that $\mathcal{A}$ cannot substitute one RAM index for another: both indexes are authentic values that cannot be be forged by $\mathcal{A}$ except with negligible probability. We emphasize that $\mathcal{A}$ also cannot "break apart" a RAM value from its index: all swap gates operate over RAM slots as a unit. That is, when $\mathcal{P}$ chooses whether or not to swap two RAM slots, that single choice will optionally swap both the value and its index. Next, we consider each swordRAM operation.

- **swordRAM initialization and additive sharings.** Consider the swordRAM initialization procedure, which permutes an array of additive secret sharings. Recall, additive secret sharings are used to encode an array of random values $K \in_\$ (\mathbb{Z}_p^s)^n$. These sharings support soundness because the shares seen by $\mathcal{A}$ are indistinguishable from uniform field elements; in order to cheat, $\mathcal{A}$ must successfully replace one index of K by another chosen index. But since each value in K is simply random, this substitution succeeds only with probability $\frac{1}{p}$. Operations over additive secret shares are sound by the same argument as for operations over authenticated secret shares.
- **swordRAM reads.** In fact, swordRAM reads are entirely local operations: $\mathcal{P}$ simply looks up her appropriate share in $(\!|\pi(K)|\!)$ and adds it to her share of $[\![A]\!]_{K[0..|A|]}$. Thus, reads are trivially sound (i.e., invalid output values cannot be authenticated).
- **swordRAM writes.** On a swordRAM write of share $a\Delta - M$, $\mathcal{V}$ sends to $\mathcal{P}$ the value $M - K[w]$ where $K[w]$ is an index of the array of random values used to mask the PrORAM content. M and $K[w]$ are both uniform field elements unknown to $\mathcal{A}$. While $K[w]$ appears elsewhere in $\mathcal{P}$'s view, it is only as part of additive shares where it is masked by a uniform mask. Thus, the value $M - K[w]$ appears indistinguishable from a random value to $\mathcal{A}$, so swordRAM writes are sound (i.e., invalid output values cannot be authenticated).

We further emphasize that each uniform mask $K[i]$ is used *exactly once*. This prevents re-using stale values or substituting one RAM value for another.

Construction 1 is sound. □

Theorem 7 (Verifiability). *Construction 1 is **verifiable**.*

Proof. By construction of a polytime algorithm `Ext`.

First, we argue that `Ve` ensures that (e, M) is properly constructed. The procedure `Ve` recovers $\mathcal{V}$'s seed from e and then replays the actions of $\mathcal{V}$ during the proof. As the verification algorithm proceeds, it compares the constructed messages with those in e and M. Thus, the verification procedure ensures that e, M were indeed correctly generated by the pseudorandom seed, and so when $\texttt{Ve}(\mathcal{C}, e, M) = 1$, the check is reliable.

Note, $\mathcal{V}$ is an interactive Turing machine, while Ve is a procedure; nevertheless, Ve (and `Ext`) can replay $\mathcal{V}$'s actions. While in our protocol presentation $\mathcal{V}$ sends messages to $\mathcal{P}$ (which ultimately constitutes $(M, \mathsf{En}(e, i))$), he receives none, other than the final message h'.

Next, we construct `Ext`. `Ext` recovers $\mathcal{V}$'s seed from e and replays $\mathcal{V}$'s actions, including computing the message d that $\mathcal{V}$ requires to accept the proof (d is the hash of $\mathcal{P}$'s opened shares, cf. Section 4.1). Thus, `Ext` has enough information to replay $\mathcal{V}$ and calculate accepting d.

Construction 1 is verifiable. $\square$