Revisiting Square-Root ORAM
Efficient Random Access in Multi-Party Computation

Samee Zahur* Xiao Wang† Mariana Raykova‡ Adrià Gascón§
Jack Doerner* David Evans* Jonathan Katz†
*University of Virginia †University of Maryland ‡Yale University §University of Edinburgh

***Abstract*—Hiding memory access patterns is required for secure computation, but remains prohibitively expensive for many interesting applications. Prior work has either developed custom algorithms that minimize the need for data-dependant memory access, or proposed the use of Oblivious RAM (ORAM) to provide a general-purpose solution. However, most ORAMs are designed for client-server scenarios, and provide only asymptotic benefits in secure computation. Even the best prior schemes show concrete benefits over naïve linear scan only for array sizes greater than 100. This immediately implies each ORAM access is 100 times slower than a single access at a known location. Even then, prior evaluations ignore the substantial initialization cost of existing schemes.**

We show how the classical square-root ORAM of Goldreich and Ostrovsky can be modified to overcome these problems, even though it is asymptotically worse than the best known schemes. Specifically, we show a design that has over $100x$ lower initialization cost, and provides benefits over linear scan for just 8 blocks of data. For all benchmark applications we tried, including Gale-Shapley stable matching and the scrypt key derivation function, our scheme outperforms alternate approaches across a wide range of parameters, often by several orders of magnitude.

I. Introduction

Over the past decade, advances in protocol design and implementation [2, 17, 26], cryptographic techniques [3, 20, 29, 41], and approaches for constructing smaller circuits [19, 31] have combined to make circuit-based secure computation efficient enough for many practical uses. Nevertheless, typical applications still exhibit an unacceptable performance penalty when computed using such protocols — especially those employing algorithms that make heavy use of data-dependent memory access. Although such accesses are constant-time operations when performed locally, they require (in general) time proportional to the size of the memory when performed using circuit-based secure-computation protocols, because the access patterns must be hidden. For this reason, researchers, beginning with Gordon et al. [15], have investigated secure computation in the *random access machine* (RAM) model of computation [7, 9, 18, 23, 24, 25, 36, 40]. A primary building block in this model is *oblivious RAM* (ORAM) [12], which provides a memory abstraction that can read and write to arbitrary locations without leaking any information about which locations were accessed.

ORAM protocols were originally proposed for a client-server setting where a client stores and manipulates an array of length n on an untrusted server without revealing the data or access patterns to the server. Gordon et al. adapted ORAM to the setting of secure computation [15], where parties collectively maintain a memory abstraction that they can jointly access access, while hiding the access patterns from everyone. In essence, the parties run a secure-computation protocol to store *shares* of the state of the underlying ORAM protocol, and then use circuit-based secure computation to execute the ORAM algorithms.

Although there is a rich literature devoted to developing ORAM protocols with improved performance [4, 13, 14, 21, 28, 30, 32, 37], most of this literature focuses on optimizing performance in the client-server setting, and most work on RAM-based secure computation (RAM-SC) uses existing ORAM protocols (to a first approximation) as black boxes. We highlight, however, that there are a number of differences between applications of ORAM in the two settings:

1) In the client-server setting the client owns the data and performs the accesses, so the privacy requirement is unilateral. In the RAM-SC setting none of the parties should be able to learn anything about the data or access patterns.
2) In the client-server setting the client's state should be sublinear in n or else the problem is trivial; for RAM-SC, however, the linear state is stored across both parties.
3) In the client-server setting the most important metric is the total communication complexity. In the

RAM-SC setting other measures of efficiency become more important. Specifically, the algorithmic complexity is important because the algorithms will be emulated using generic secure computation.

4) In the client-server setting, the initialization step (when the client outsources its data to the server) is "free" because it is a local action on the part of the client. In the RAM-SC case, the parties must use a distributed protocol for initialization and the cost of doing so may be prohibitive.

Existing work on ORAM has focused entirely on *asymptotic* performance; we are not aware of any prior work whose aim is to improve performance for concrete values of n. Indeed, prior work in the RAM-SC setting has found that a linear scan over the data (i.e., a trivial ORAM construction) outperforms more-complicated ORAM constructions until n becomes quite large [15, 34, 35] (in practice, n is often small even when the inputs are large since n may denote the length of a single array rather than the entire memory being used by the computation, and each memory block may contain many individual data items). This means that for practical sizes, the entire body of research on ORAM has had little impact as far as RAM-SC is concerned.

Contributions. We re-visit the classical square-root ORAM of Goldreich and Ostrovsky [12], and propose a number of modifications to that construction with the goal of obtaining an ORAM scheme suitable for secure computation in the semi-honest setting:

1) In the original scheme, the client uses a hash function to compute the *position map* (i.e., the mapping from semantic addresses to physical addresses). We replace this with a shared array storing the position map explicitly. This is particularly beneficial when the underlying ORAM algorithms are computed using generic circuit-based secure computation.
2) Because the position map is stored explicitly, initialization and reshuffling (expensive operations performed sporadically) can be made much more efficient than in the original construction, as they can be based upon Waksman shuffling networks [33] rather than oblivious sorting.
3) As observed in prior work [32] the position map is a constant factor smaller than the original memory array. We use ORAMs recursively to enable oblivious access to the position map, and develop a number of optimizations in order to obtain a secure and efficient protocol.

We implement and evaluate our construction (code available at http://oblivc.org/) and show that for small-to-moderate values of n our scheme offers more efficient data access than Circuit ORAM [34]. In fact, our scheme outperforms even the trivial ORAM (i.e., linear scan) for n as small as 8 (the exact crossover point depends on the block size used as well as the underlying network and processor). Our construction also significantly outperforms prior work in terms of initialization time. To understand how the properties of different applications impact ORAM performance, and demonstrate the general applicability of our design, we implement and evaluate several benchmark application, including secure two-party computations of the Gale-Shapley stable matching algorithm, breadth-first search, binary search, and the Scrypt hash function. The resulting protocols are more efficient than prior approaches by an order of magnitude or more in some cases.

II. Background

This section provides a brief introduction to multi-party computation (MPC), oblivious RAM (ORAM), RAM-based secure computation (RAM-SC), and closely related protocols.

A. Multi-Party Computation

Secure multi-party computation [11, 38] enables two or more parties to collaboratively evaluate a function that depends on secret inputs from all parties, while revealing nothing but only the result of the function. In most generic constructions of multi-party computation, the function to be evaluated is represented as a circuit (either Boolean or arithmetic). Numerous circuit-based multi-party computation protocols have been developed for different scenarios. In this paper, we focus on using our ORAM design with Yao's garbled circuit protocol. However, our scheme also works with other MPC protocols in general, and does not depend on any specifics of Yao's protocol.

Garbled circuits protocols involve parties, denoted the *generator* and *evaluator*. Given a publicly known function f, the generator associates each input bit with two garbled keys k^0, k^1, and computes a "garbled" circuit representation of the function f, GC_f. Given garbled keys corresponding to inputs x and y, the evaluator can obliviously evaluate GC_f to learn garbled keys for output $f(x,y)$. The generator generates and sends GC_f and the input keys for its own input. The generator and evaluator execute an oblivious transfer protocol to enable the evaluator to learn the input keys corresponding to its input without revealing that input to the generator. After

obtaining its input keys, the evaluator can obliviously evaluate GC_f to obtain the output keys which are decoded in the final step.

B. Oblivious RAM

Oblivious RAM provides a memory structure that hides access patterns [12]. An ORAM scheme consists of two protocols: an *initialization protocol* that takes as input an array of elements, initializes a new oblivious structure in memory; and an *access protocol* that implements each logical access to the ORAM with a sequence of physical accesses to the underlying structure.

To be secure, an ORAM must satisfy two properties: 1) the physical access pattern of the initialization protocol is indistinguishable when initializing different input arrays of the same size; 2) for any two sequences of semantic accesses of the same length, the physical access patterns produced by the access protocol must be indistinguishable. Note that it is always possible to implement a secure initialization protocol by performing the access protocol iteratively on all input elements, and this is the approach taken by previous ORAM designs used in RAM-SC. It can be, however, very inefficient to initialize the ORAM through repeated accesses.

Goldreich and Ostrovsky [12] introduced two ORAM constructions with a hierarchical layered structure: the first, *Square-Root ORAM*, provides square root access complexity; the second, *Hierarchical ORAM*, requires a logarithmic number of layers and has polylogarithmic access complexity. A recent series of ORAM schemes, beginning with the work of Shi et al. [30], adopted a sequence of binary trees as the underlying structure. While, asymptotically, the most bandwidth efficient ORAM constructions known use the hierarchical paradigm [21], tree-based ORAMs are considered more efficient for practical implementations especially when used in MPC protocols. This is primarily because classical hierarchical constructions use hash functions or pseudorandom functions (PRFs) to shuffle data in the oblivious memory. In an MPC context these functions must be executed as secure computations with large circuits.

C. RAM-Based Secure Computation

In traditional MPC, general input-dependent array access incurs a linear-time overhead since all elements need to be touched to hide the position of interest. RAM-based secure computation (RAM-SC) combines ORAMs with circuit-based MPC protocols, to enable secure random memory accesses [15]. In RAM-SC, the bulk of the computation is still performed by a circuit-based protocol as in traditional MPC, but memory accesses are performed using an ORAM that is implemented within the MPC protocol. For each access, the circuit now emulates an ORAM access step to translate a secret logical location into multiple physical locations that must be accessed. The physical locations are then revealed to the two parties, which pass the requested elements back into the circuit for use in the oblivious computation. Finally, the circuit produces new data elements to be written back to those physical positions, hiding which elements were modified and how they were permuted. One such structure is maintained for each array that needs input-dependent general random access.

Two-party RAM-SC was first formulated by Gordon et al. [15] with an implementation based on a tree-based ORAM scheme proposed by Shi et al. [30]. Subsequent works [7, 9, 18, 36] presented improved protocols, all based on tree-based ORAM constructions. Wang et al. [34] proposed Circuit ORAM, which yields the best known circuit size both in terms of asymptotic behavior and concrete performance. In Section V, we provide performance comparisons between our new ORAM scheme and Circuit ORAM, showing orders of magnitude improvement for access and initialization across a wide range of parameters and applications.

D. Variations

In addition to the RAM-SC model we focus on, there are other uses for ORAMs in secure computation protocols. Some of the ORAM innovations produced in these settings have been applied to the RAM-SC designs in Section II-C. Although it is beyond the scope of this work, we believe our ORAM design may likewise yield benefits in other contexts.

Gentry et al. [9] proposed several optimizations for tree-based ORAMs and considered briefly how to build a HE-over-ORAM system. A system based on Path ORAM [32] was built in their subsequent work [10]. They showed a per-access time of 30 minutes for a database with 4 million 120-bit records, excluding the cost of initialization.

Lu and Ostrovsky [25] designed an ORAM algorithm based on two non-colluding servers. When applied to a two-party secure RAM computation setting, these servers become parties engaging in an MPC protocol. Their construction achieves $O(\log N)$ overhead, but suffers from huge concrete costs because it requires oblivious evaluation of $\Theta(\log N)$ cryptographic operations per access, which is prohibitively expensive in an MPC protocol.

Afshar et al. [1] discussed how to extend RAM-SC with malicious security, where both parties can arbitrarily deviate from the protocol. They proposed efficient consistency checks that avoid evaluating MAC in circuits. In this paper, we only consider semi-honest adversaries, and hope that future work will extend our protocol to be secure against malicious adversaries.

III. Revisiting Square-Root ORAM

In this section we revisit Goldreich and Ostrovsky's square-root ORAM design [12] and adapt it to the RAM-SC setting. Section III-A introduces notations used to describe ORAM algorithms; Section III-B provides a brief description of the original scheme; Section III-C introduces a basic (but inefficient) construction by making some key changes to the original scheme; Section III-D shows how to improve its efficiency with a recursive construction which is our final design.

A. Notation

We use $\langle x \rangle$ to denote a variable x secretly shared by the two parties. In our garbled circuit implementation, $\langle x \rangle$ means the generator knows (k^0, k^1) and the evaluator knows k^x. Since the actual value of x is not known to either party, we interchangeably use the terms "private", "garbled", and "oblivious" to describe it.

The length of an array is always public, although padding can be used to hide its exact length when necessary. An array containing private elements is denoted using angle brackets (e.g., $\langle \mathsf{Array} \rangle$). We denote the i^{th} element of an array using a subscript (e.g., $\langle \mathsf{Array} \rangle_i$). The index may be oblivious (e.g., $\langle \mathsf{Array} \rangle_{\langle i \rangle}$), in which case the array access is performed via linear scan.

The structure blocks represents an array of block objects. Each block contains private data, $\mathsf{block}.\langle \mathsf{data} \rangle$, and a private record of its logical index, $\mathsf{block}.\langle \mathsf{index} \rangle$. Thus, i is the physical index of blocks_i, and $\mathsf{blocks}_i.\langle \mathsf{index} \rangle$ is the logical index of the same block. Neither changing the contents of a block nor moving it from one structure to another alters its logical index, unless explicitly noted.

In pseudocode, ordinary conditional statements will use the keyword **if**, while conditionals on secret values will use $\langle$**if**$\rangle$. The bodies of secret conditionals are always executed, but the statements in them are executed conditionally, becoming no-ops if the condition is false.

We use $\langle a \rangle \xleftarrow{\$} B$ to denote random choice of a secret element a from a public set B.

Figure 1 shows how the access algorithm for a naïve linear scan ORAM is written in our notation. The algorithm Access takes three parameters as inputs:

- Oram: the main data structure storing the payload.
- $\langle i \rangle$: the private, logical index of the block we want to access.
- $\Phi()$: a function that is invoked during access to read, write or modify the desired block.

The ORAM hides index $\langle i \rangle$ by performing a linear scan over all elements. Note that we use $\langle$**if**$\rangle$ for the conditional, so the body of the conditional statement will actually be executed n times, although only one will take effect. Both parties will see the garbled keys representing $\langle \mathsf{val} \rangle$ or $\langle \mathsf{data} \rangle$ change n times inside $\Phi()$; they just won't know if the associated plaintext has also changed, since that depends on secret index $\langle i \rangle$.

Users will not typically use ORAMs by directly invoking Access, but by using the wrapper functions shown in Figure 2. These wrappers are exactly the same across all ORAM constructions we consider; the essential logic is in Access.

```
define Access(Oram, ⟨i⟩, Φ):
  for j from 0 to Oram.n − 1:
    ⟨if⟩ ⟨i⟩ = j: Φ(Oram_j)
```

Fig. 1: Access **algorithm for the linear scan ORAM.**

```
define Write(Oram, ⟨i⟩, ⟨val⟩):
  define Φ(block):
    block.⟨data⟩ ← ⟨val⟩
  Access(Oram, ⟨i⟩, Φ)

define Read(Oram, ⟨i⟩):
  ⟨val⟩ ← ⊥
  define Φ(block):
    ⟨val⟩ ← block.⟨data⟩
  Access(Oram, ⟨i⟩, Φ)
  return ⟨val⟩
```

Fig. 2: Read and write wrappers defined using Access()

B. Square-Root ORAM

Figure 3 shows the original square-root ORAM proposed by Goldreich and Ostrovsky [12]. The ORAM structure consists of following components:

1) Shuffle: an array of blocks, also referred to as "permuted memory" in the original paper.

```
define Initialize(blocks, T)
  n ← |blocks|
  ⟨π⟩ ← pseudorandom function
  append √n dummy blocks to Shuffle
  Shuffle ← ObliviousSort(blocks, ⟨π⟩)
  Oram ← (n, t ← 0, T, ⟨π⟩, Shuffle, Stash ← ∅)
  return Oram

define Access(Oram, ⟨i⟩, Φ)
  ⟨found⟩ ← false
  for j from 0 to Oram.t:
    ⟨if⟩ Oram.Stash_j.⟨index⟩ = ⟨i⟩:
      ⟨found⟩ ← true
      Φ(Oram.Stash_j)
  ⟨if⟩⟨found⟩ :
    ⟨k⟩ ← Oram.n + Oram.t
  ⟨else⟩ : ⟨k⟩ ← ⟨i⟩
  p ← reveal(π(⟨k⟩))
  ⟨if⟩ not ⟨found⟩ :
    Φ(Oram.Shuffle_p)
  append Oram.Shuffle_p to Oram.Stash
  Oram.Shuffle_p ← dummy
  Oram.t ← Oram.t + 1
  if Oram.t = Oram.T:
    blocks ← real blocks in
      Oram.Shuffle ∪ Oram.Stash
    Oram ← Initialize(blocks, Oram.T)
```

Fig. 3: **The original square-root ORAM scheme [12].**

```
define Initialize(blocks, T)
  n ← |blocks|
  ⟨π⟩ ← random permutation on n elements
  Shuffle ← ObliviousPermute(blocks, ⟨π⟩)
  Oram ← (n, t ← 0, T, ⟨π⟩, Shuffle,
          Used ← ∅ , Stash ← ∅)
  return Oram

define Access(Oram, ⟨i⟩, Φ)
  ⟨found⟩ ← false
  for j from 0 to Oram.t:
    ⟨if⟩ Oram.Stash_j.⟨index⟩ = ⟨i⟩:
      ⟨found⟩ ← true
      Φ(Oram.Stash_j)
 ⟨if⟩⟨found⟩ :
    ⟨p⟩ ←$ {0, ..., (Oram.n − 1)} \ Oram.Used
  ⟨else⟩ : ⟨p⟩ ← Oram.⟨π⟩_⟨i⟩
  p ← reveal(⟨p⟩)
  ⟨if⟩ not ⟨found⟩ :
    Φ(Oram.Shuffle_p)
  append Oram.Shuffle_p to Oram.Stash
  Oram.Used ← Oram.Used ∪ {p}
  Oram.t ← Oram.t + 1
  if Oram.t = Oram.T:
    for j from 0 to |Oram.Used| − 1:
      p′ ← Oram.Used_j
      Oram.Shuffle_p′ ← Oram.Stash_j
    Oram ← Initialize(Oram.Shuffle, Oram.T)
```

Fig. 4: **Our basic square-root ORAM scheme.**

2) Stash: an array of blocks, termed the "shelter" in the original paper.
3) π: a pseudorandom function (PRF) mapping indices to random strings. Note that π needs to be evaluated securely using MPC protocols, which is why previous RAM-SC designs dismissed the square-root ORAM construction.

To initialize an ORAM from an array of blocks, we first append $\sqrt{n}$ dummy blocks to the input array and obliviously permute all $n+\sqrt{n}$ blocks according to the pseudorandom permutation $\pi(i)$. Once the blocks are shuffled, their physical locations and semantic indices are uncorrelated, and so each block may be accessed once (and only once) without revealing anything about the access pattern. But, if a location in the shuffled array is accessed multiple times that would leak information, revealing that the access sequence contains a repeated access.

To access logical index $\langle i \rangle$, we first linear scan the blocks in Stash. If the block is not found, we compute its physical location, $\langle p \rangle \leftarrow \pi(\langle i \rangle)$; otherwise, we find the physical location of the next unaccessed dummy blocks, $\langle p \rangle \leftarrow \pi(n + \mathsf{Oram}.t)$. The value $\langle p \rangle$ is then revealed to both parties, but leaks no information about the logical index. The block at the physical location referred to by p is accessed, by doing a binary search over the $\pi(i)$

values, and moved to Stash.

After T accesses have been performed, we obliviously remove all dummy blocks in Oram.Stash and Oram.Shuffle and re-initialize the whole structure. The complete protocol is shown in pseudocode in Figure 3. In Goldreich and Ostrovsky's original protocol, Oram.T is set to $\sqrt{n}$, resulting in $O(\sqrt{n}\log_2 n)$ amortized cost per access.

The original scheme was not designed for a RAM-SC setting, and suffers from two key problems that make it very expensive to implement in an MPC:

1) It evaluates the PRF $\pi(x)$ for each access; in the initialization algorithm, $n+\sqrt{n}$ evaluations of $\pi(i)$ are needed. This is inefficient, especially in MPC protocols since evaluating each PRF requires tens of thousands of gates.
2) It requires a $\Theta(n\log^2 n)$ oblivious sort on the data blocks in two different places: to shuffle data blocks according to the PRF results, and to remove dummy blocks before initialization.

Next, we discuss how to adapt the scheme for efficient use in RAM-SC by eliminating these problems.

C. Basic Construction

Figure 4 presents our basic construction, a step towards our final scheme. The construction is similar to the original scheme, with a key difference: instead of using PRF to generate a random permutation, it stores the permutation π explicitly as a private array. This enables several performance improvements:

1) Storing the permutation π as a private array enables us to replace oblivious sorting during the initialization with a faster oblivious permutation. In addition, the value p revealed during the access refers to the real location, which avoids using binary search to find the location for p. Section III-D shows how to recursively implement π for better efficiency.
2) We eliminate the need of dummy blocks. When a dummy access is needed, we instead access a random location for real blocks that is not accessed before and append the block to the Stash.
3) By using a public set Used, we avoid the oblivious sorting needed when moving blocks from the Stash to Shuffle. This is efficient since Used is maintained in the clear and is secure because all elements in Used have already been revealed.

Security. Assuming the MPC protocol itself is secure and does not leak any information about oblivious variables, this protocol satisfies the ORAM requirement that no information is revealed about the logical access pattern. On each access, a uniform unused element from Shuffle is selected, regardless of the semantic index requested. Subsequently, the entire Stash is always scanned. Finally, the entire structure is reshuffled at a fixed interval, in a manner independent of the access pattern. The only values revealed are the permuted physical indices p; the set Used, which contains no information about the semantic indices; and the counter t, which increments deterministically.

Asymptotic cost. Now we analyze the average cost of accessing a block in this basic scheme. We represent the combined cost of accessing $\langle\pi\rangle$ and Used as $c(n)$, some value that only depends on the number of blocks, n, but not block size. We use B to denote the cost of accessing a single block (this could be bandwidth, time, or energy cost). The augmented cost, $B' = B + \Theta(\log_2 n)$, includes the additional cost of accessing the metadata containing the block's logical index. For an ORAM of size n, each logical index requires $\log_2 n$ bits, so it incurs $\Theta(\log_2 n)$ cost to retrieve or compare an index.

Since our construction is a periodic algorithm that performs a shuffle every T accesses, we obtain the amortized per-access cost by computing the average over T accesses. This is the cost of the shuffle plus the cost of B' for each block touched thereafter until the next shuffle.

The cost of shuffling is approximately $B'W(n)$ using a Waksman network [33]. Here, $W(n) = n\log_2 n - n + 1$ is the number of oblivious swaps required to permute n elements. On each access, the entire Stash, comprising t blocks, must be scanned. Thus, the total cost of the T accesses and one shuffle which constitutes a full cycle is given by

$$\begin{aligned} & B'W(n) + \sum_{t=1}^{T} \left(B't + c(n)\right) \\ \leq\ & B'n\log_2 n + \frac{1}{2}B'T(T+1) + Tc(n) \\ =\ & T\left(\frac{1}{T}B'n\log_2 n + \frac{1}{2}B'(T+1) + c(n)\right) \\ =\ & TF(n) \end{aligned}$$

where $F(n)$ is the amortized per-access cost we are after.

If reshuffle period $T = \Theta(\sqrt{n\log_2 n})$, the asymptotic cost is $F(n) = \Theta(B'\sqrt{n\log_2 n})$, assuming the block size is large enough to make $c(n)$ negligible compared to B.

Concrete cost. This design is less expensive than linear scan, even for reasonably small block sizes and for block

```
define Initialize(blocks)
  n ← |blocks|
  ⟨π⟩ ← random permutation on n elements
  Shuffle ← ObliviousPermute(blocks, ⟨π⟩)
  T ← ⌈√W(n)⌉
  Oram_1 ← InitializePosMap(⟨π⟩, 1, T)
  Oram_0 ← (n, t ← 0, T, Oram_1, Shuffle,
            Used ← ∅, Stash ← ∅)
  return Oram_0
```

```
define Access(Oram_0, ⟨i⟩, Φ, )
  ⟨found⟩ ← false
  for j from 0 to Oram.t:
    ⟨if⟩ Oram_0.Stash_j.⟨index⟩ = ⟨i⟩:
      ⟨found⟩ ← true
      Φ(Oram_0.Stash_j)
  p ← GetPos(Oram_0.Oram_1, ⟨i⟩, ⟨found⟩)
  ⟨if⟩ not ⟨found⟩:
    Φ(Oram_0.Shuffle_p)
  append Oram_0.Shuffle_p to Oram_0.Stash
  Oram_0.Used ← Oram_0.Used ∪ {p}
  Oram_0.t ← Oram_0.t + 1
  if Oram_0.t = Oram_0.T:
    for j from 0 to Oram_0.T − 1:
      p′ ← Oram.Used_j
      Oram_0.Shuffle_p′ ← Oram_0.Stash_j
    Oram_0 ← Initialize(Oram_0.Shuffle)
```

Fig. 5: **Our recursive square-root ORAM scheme.** $W(n)$ is the number of swaps needed in a n-sized Waksman permutation network.

counts as low as four. With linear scan, the cost is nB per access, ignoring smaller terms that are independent of B. With four blocks, the cost of a linear scan is $4B$. Using a shuffling period of $T = 3$, we get a cost of $B(W(4)+1+2+3) = 11B$ for three accesses, again ignoring smaller terms that are independent of B. This is slightly better than the linear scan cost for three accesses, $3 \times 4B = 12B$. Thus, for four blocks of a large enough size, the simplified one-level square-root ORAM is less expensive than a linear scan, even after accounting for the cost of initialization. However, in the case of small blocks, the terms independent of B (which we have ignored) become significant enough that linear scan has a slight advantage.

In our experiments, we observed the square-root scheme to be more efficient in terms of bandwidth for four blocks of just 36 bytes each (see Section V-B for details). For larger block sizes, we found that the cost ratio reaches $11:12$, as expected.

D. Scalable Construction

So far, we have not discussed how to implement the structure $\langle\pi\rangle$ more efficiently than linear scan, aside from claiming that its costs do not depend on the block size. For small values of n, linear scan is good enough, as in the four-block example above. At this size, π comprises just four records of two secret bits each. However, for larger values of n, it may seem natural to build these structures upon recursive ORAMs of decreasing size. As we discuss next, however, this method is unacceptably costly. Our solution is to specialize the structure for position maps.

The position map structure, $\langle\pi\rangle$, is common to most existing tree-based constructions [30, 32, 34]. It is usually implemented atop recursive ORAMs of decreasing size, each level packing multiple indices of the previous into a single block, and the whole thing is updated incrementally as elements of the main ORAM are accessed. In these constructions, each ORAM lookup requires a single corresponding lookup in each recursive position maps. However, in our scheme, a naïve recursive structure for $\langle\pi\rangle$ would require $n+T$ position lookups for every T accesses to the main ORAM (where T is the number of accesses between shuffles) since each of the T main accesses would require an access to the position map, and additional n accesses would be required to store the regenerated permutation π' when the ORAM is shuffled.

This is a serious problem: each level of the recursive structure would need to store pack indices of the previous level in a single block, which would be traversed by linear scan. Thus, each subsequent level decreases in element count by a factor of pack, but all levels require pack time to linear scan the relevant block. We can multiply by $(n+T)/T$ to amortize the cost over T accesses, where $T = \sqrt{n \log_2 n}$, the shuffle period (as

```
define InitializePosMap(⟨π⟩, k, T)
  n ← |⟨π⟩|
  if n/pack ≤ T:
    ⟨Used_{0...(n−1)}⟩ ← (false, ..., false)
    Oram_k ← (n, t ← 0, T, ⟨π⟩, ⟨Used⟩)
  else:
    for i ∈ {0 ... ⌈n/pack⌉ − 1}:
      ⟨data⟩ ← (⟨π⟩_{pack·i}, ..., ⟨π⟩_{pack·(i+1)−1})
      blocks_i ← (⟨data⟩, ⟨index⟩ ← i)
    ⟨π′⟩ ← random permutation on ⌈n/pack⌉ elements
    Shuffle ← ObliviousPermute(blocks, ⟨π′⟩)
    Oram_{k+1} ← InitializePosMap(⟨π′⟩, k+1, T)
    Oram_k ← (n, t ← 0, T, Oram_{k+1}, Shuffle,
              Stash ← ∅)
  return Oram_k

define GetPosBase(Oram_k, ⟨i⟩, ⟨fake⟩)
  ⟨p⟩ ← ⊥
  ⟨done⟩ ← false
  for j from 0 to (Oram_k.n − 1):
    ⟨s_1⟩ ← (not ⟨fake⟩ and ⟨i⟩ = j)
    ⟨s_2⟩ ← (⟨fake⟩ and not Oram_k.⟨Used⟩_j
                and not ⟨done⟩)
    ⟨if⟩ ⟨s_1⟩ or ⟨s_2⟩:
      ⟨p⟩ ← ⟨π_j⟩
      Oram_k.⟨Used⟩_j ← true
      ⟨done⟩ ← true
  p ← reveal(⟨p⟩)
  return p

define GetPos(Oram_k, ⟨i⟩, ⟨fake⟩)
  if Oram_k.n/pack ≤ Oram_k.T:
    p ← GetPosBase(Oram_k, ⟨i⟩, ⟨fake⟩)
  else:
    ⟨found⟩ ← false
    ⟨h⟩ ← ⟨i⟩/pack
    ⟨l⟩ ← (⟨i⟩ mod pack)
    for j from 0 to Oram_k.t − 1:
      ⟨if⟩ Oram_k.Stash_j.⟨index⟩ = ⟨h⟩:
        ⟨found⟩ ← true
        block ← Oram_k.Stash_j
        ⟨p⟩ ← block.⟨data⟩_⟨l⟩
    p′ ← GetPos(Oram_{k+1}, ⟨h⟩, ⟨fake⟩ or ⟨found⟩)
    append Oram_k.Shuffle_{p′} to Oram_k.Stash
    Oram_k.t ← Oram_k.t + 1
    ⟨if⟩ ⟨fake⟩ or not ⟨found⟩:
      block ← Oram_k.Stash_{(Oram_k.t−1)}
      ⟨p⟩ ← block.⟨data⟩_⟨l⟩
    p ← reveal(⟨p⟩)
  return p
```

Fig. 6: **Implementation of the recursive position map.**

computed in Section III-C). If the amortized cost per access to level i of this map is $c_i(n)$, we have:

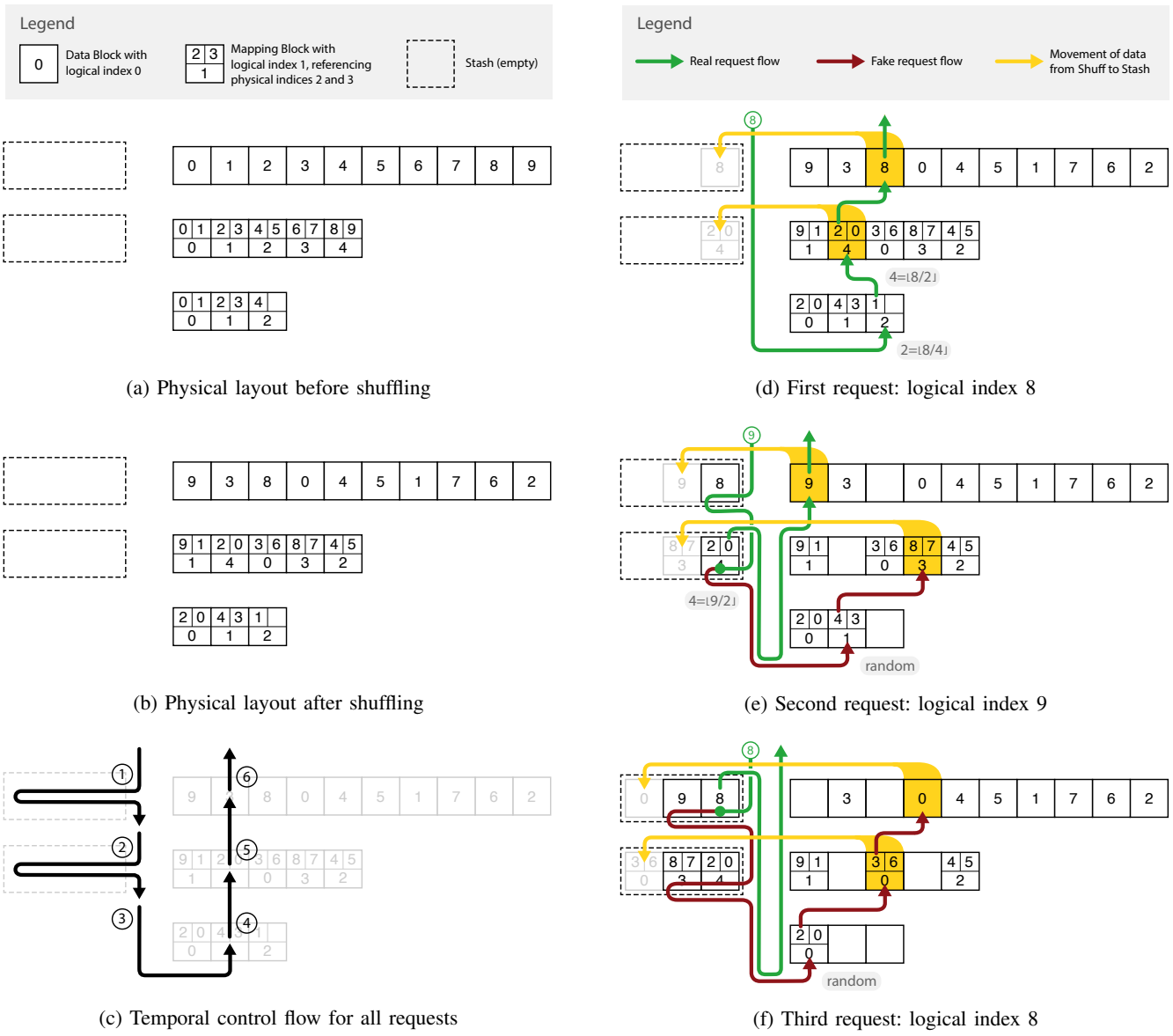
$$\begin{aligned} c_i(n) &\geq \frac{n+T}{T}(c_{i+1}(n/\mathsf{pack}) + \mathsf{pack}) \\ &\geq \frac{n}{\sqrt{n\log_2 n}} c_{i+1}(n/\mathsf{pack}) \\ &\geq \sqrt{\frac{n}{\log_2 n}} c_{i+1}(n/\mathsf{pack}). \end{aligned}$$

This is a super-polynomial function with $\Theta(\log n)$ levels of recursion, which is unacceptable for our efficiency goals. Fixing this involves three changes to our basic construction.

The first change is to take advantage of our ability to initialize quickly from an oblivious array. On each shuffle, we regenerate π, and, instead of writing it into the recursive structure element by element, we re-initialize the recursive structure using π' as the seed data. This eliminates the extra n accesses to the position map on each cycle.

Second, we lock all levels of the recursive structure to the same shuffle period, $T = \sqrt{n\log_2 n}$, where n is the number of blocks in the main ORAM (the level that contains the original data). We terminate the recursion at the first level with fewer than T blocks, and access this final level using linear scan. Using this arrangement, we can initialize the entire ORAM in $\Theta(Bn\log n)$ bandwidth and time.

This second modification has a downside. All levels of the recursive ORAM shuffle in synchronization with one another, based on a shuffle period determined by the largest level. This shuffle period will be significantly



(a) Physical layout before shuffling

(b) Physical layout after shuffling

(c) Temporal control flow for all requests

(d) First request: logical index 8

(e) Second request: logical index 9

(f) Third request: logical index 8

Fig. 7: **Illustration of data flow for one full cycle of an example ORAM.** In subfigures (d), (e), and (f) we present the logical dependencies for three sequential accesses.

suboptimal for levels with fewer blocks. We pay a time and bandwidth cost of $\Theta(T)$ at each level of the ORAM (for linearly scanning the $\Theta(T)$ blocks in each level's Stash). An ORAM instantiated with n elements will have $\log n$ levels, which brings the cost per access to $\Theta(T \log n) = \Theta(\sqrt{n \log^3 n})$. However, the linear scan overhead incurred by using a global shuffling period is compensated for by gains in the efficiency of Used which it enables.

Constructing an efficient mechanism for keeping track of used and unused physical blocks poses a challenge. Used contains inherently public data — both parties are aware which physical locations in Shuffle have already been accessed — yet, they must obliviously check whether it contains a secret logical index $\langle p \rangle$. Moreover, they must be able to sample a secret, uniform element from $S = \{0, \ldots, n-1\} \setminus \text{Used}$. The simplest method would be to sample an integer from $\{0, \ldots, |S|-1\}$ and then obliviously map it to the set S, an expensive operation.

The third change removes the need to obliviously check Used for secret index $\langle p \rangle$. Instead of using an

explicit data structure, our choice of a global shuffle period allows us to implicitly represent a superset of Used in the recursive structure $\langle\pi\rangle$, by tracking which blocks the smallest recursive level have been used. We use the notation Oram_k.Stash, Oram_k.Shuffle, and Oram_k.Used to represent the corresponding structures in recursive ORAM at level k. Oram_0 is the main ORAM that holds the data blocks; Oram_1 is the top level of the position map $\langle\pi\rangle$; Oram_2 and so on indicate deeper levels of the recursive position map structure.

We maintain the invariant that if a block has already been moved from Oram_n.Shuffle to Oram_n.Stash, the corresponding block in Oram_{n+1} has also been moved from Oram_{n+1}.Shuffle to Oram_{n+1}.Stash. The converse is not necessarily true: it is possible for Oram_{n+1}.Stash to contain blocks that map to unaccessed blocks in Oram_n. This can happen, for example, if logical block i of Oram_0 has been accessed and block $i+1$ has not, but mapping information for both blocks resides in the same block of Oram_1.

Randomly sampling an unused block with this construction is simple. At the smallest level the blocks are linearly scanned, so we just pick the first unused element. This is guaranteed to point to a random unused position. At the next recursive level, we can use any element in the block referred to by the index from the first level, since they are all random and unused. The process continues to ripple upward until an unused block in the required ORAM level has been selected. This method excludes from the set to be randomly sampled any block referred to by a block that has been accessed at a lower level. Nonetheless, blocks sampled randomly remain indistinguishable from genuine accesses, as, for each top level access, exactly one unused block is accessed at each lower level.

The final construction is presented in Figures 5 and 6, and the life-cycle of the ORAM is illustrated in Figure 7.

IV. Techniques and Optimizations

This section presents some of the lower-level techniques used in our implementation.

Shuffling. We employ a Waksman network [33] for shuffling. The network executes many oblivious swap operations, each controlled by a secret bit determined by the permutation π. Let B be the number of bytes transferred when obliviously swapping two blocks of data. Since a Waksman network for shuffling requires $W(n) = n\log_2 n - n + 1$ swap operations, it is expected that the two parties will transfer $BW(n)$ bytes during a shuffle, excluding the secret control bits.

The control bits pose a problem: neither party can learn anything about the randomly sampled permutation π, but we do not know an efficient oblivious algorithm for computing the corresponding control bits. To solve this problem, we perform two shuffles: the parties locally a pick secret permutation each and compute their corresponding control bits in the clear. Each party's local permutation constitutes its share in the final secret permutation π, which is the composition of the two permutations. So long as at least one party behaves honestly, the result is a uniformly random permutation, discoverable to neither. They can jointly shuffle the data by running two consecutive shuffling networks, one for each permutation.

Performing a shuffle in this way is quite inexpensive. The bandwidth cost of $2W(n)$ swaps is comparable to $W(n)$ AND gates, using the oblivious shuffle design from Huang et al. [16] and half-gates technique from Zahur et al. [41]. However, each time we perform a shuffle, we incur the latency of a network round-trip, since the evaluator retrieves new garbled labels for control bits via oblivious transfer extension [2].

Computing the permutation. Whenever the data in Shuffle is shuffled, we must reinitialize the recursive position map so that it contains the new secret permutation, π. The first time we perform a shuffle obliviously computing π is straightforward. Because the shuffle was performed with the composition of two Waksman networks as described previously, we can run the same network backwards using $(0,\ldots,n-1)$ as inputs to obtain π.

On subsequent shuffles, the process becomes complicated. The blocks in Shuffle are no longer in logical order because they have previously been shuffled and moved from Shuffle to Stash and back. Obtaining the permutation by the same method as above would require us to run both shuffles (four Waksman networks in total) in reverse, along with any other swaps that may have happened due to ordinary ORAM access. Each additional shuffle requires two more Waksman networks, and the number continues to increase without bound.

Instead, we augment each data block with a secret record of its logical index. When the blocks are shuffled, the logical indices are shuffled with them through the Waksman networks, and these indices comprise π^{-1}, the mapping from physical to logical index. To find π, the mapping from logical to physical, we simply invert π^{-1}.

To invert π^{-1} efficiently without allowing either party to learn anything about it, we adopt a technique from

Damgård et al. [6]. The first party (Alice) locally samples a new random permutation π_a and computes the corresponding Waksman control bits. This is then used to jointly permute the elements of the secret permutation π^{-1}, producing $\pi^{-1} \cdot \pi_a = \pi_b$. Next, π_b is revealed to the second party, Bob (but not to Alice). Bob does not learn anything about π^{-1} because it is masked by π_a. Bob now locally computes π_b^{-1}, and the two parties jointly execute another Waksman network to compute $\pi_a \cdot \pi_b^{-1} = \pi$.

V. Evaluation

To evaluate our design, we implemented our Square-Root ORAM design and Circuit ORAM, the best-performing previous ORAM design, using the same state-of-the-art MPC frameworks, and measured their performance on a set of microbenchmarks. We also wanted to understand the impact of different ORAM designs on application performance, and how close we are to enabling general-purpose MPC. To this end, we implemented several application benchmarks representing a wide range of memory behaviors and evaluated their performance with different ORAM designs.

A. Experimental Setup

We implemented and benchmarked RAM-SC protocols based on our ORAM as well as Circuit ORAM, using the Obliv-C [40] framework executing a Yao's garbled circuit protocol. Obliv-C provides a C-like language interface, and it incorporates many recent optimizations [3, 17, 41].

All code was compiled using gcc version 4.8.4, with the -O3 flag enabled. Unless otherwise specified, all reported times are wall-clock time for the entire protocol execution. Our benchmarks were performed with commercially available computing resources from Amazon Elastic Compute Cloud (EC2). We used compute-optimized instances of type C4.2xlarge running Amazon's distribution of Ubuntu 14.04 (64 bit). These notes provide four physical cores (capable of executing eight simultaneous threads in total), partitioned from an Intel Xeon E5-2666 v3, and 15 GiB of memory Our benchmarks are all single-threaded and cannot saturate the processing power available. We selected C4.2xlarge nodes on the basis of the greater bandwidth and memory they offer. Each benchmark was executed between two separate nodes within the same datacenter. We used iperf to measure the inter-node bandwidth, and found it to be about 1.03 Gbps.

In addition to square-root ORAM, we benchmarked a simple linear scan and an implementation of Circuit ORAM, the best previously reported ORAM construction for MPC. Our implementation of Circuit ORAM is much more efficient than the original implementation described in Wang et al. [34]. For example, while executing benchmarks on an Amazon C4.8xlarge EC2 instance for an ORAM of one million 32-bit blocks, they reported an access time of two seconds. On a less powerful, more bandwidth-constrained C4.2xlarge EC2 instance, our implementation requires only 0.16 seconds per access for an ORAM with the same parameters. This reduction by a factor of roughly twelve is mostly due to the efficiency advantages of the Obliv-C framework over the ObliVM [24] framework used by Wang et al.'s implementation. For all performance reported in the following, we let Circuit ORAM and square-root ORAM pack 8 entries in each recursive level. Circuit ORAM stops recursion when there are fewer than 2^8 entries.

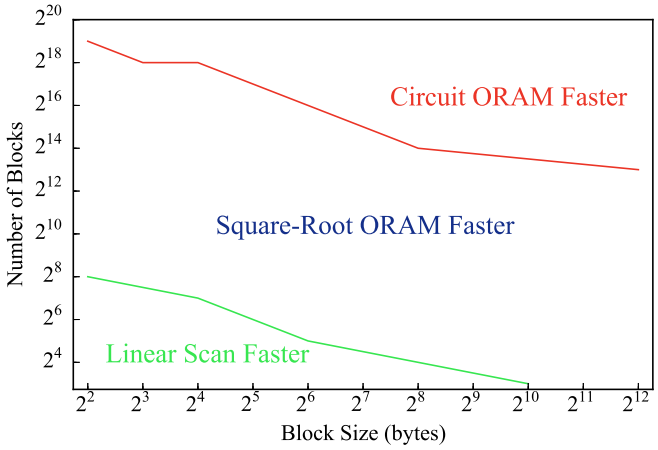


Fig. 8: **Per-access cost crossover points between ORAM schemes.** Below the green line, linear scan is most efficient. Above the red line, Circuit ORAM is most efficient. Between the two, Square-Root ORAM is most efficient.

B. Microbenchmarks

We performed several microbenchmarks to assess the granular performance of different ORAM designs. We observed single-access execution time for block counts varying from 4 to 1024 and block sizes varying from 4 to 1024 bytes. This is the region of parameter space where the efficiencies of Square-Root ORAM and linear scan overlap. Figure 8 shows the efficiency crossover points derived from this data, ignoring initialization cost. Due to the nature of the Square-Root ORAM algorithm, each access is more expensive than the previous one, until a shuffle occurs and resets the cycle. To ensure our

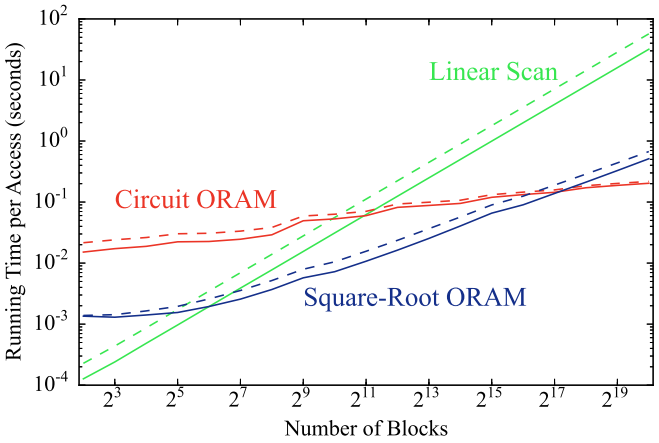


Fig. 9: **Cost per access omitting initialization.** Solid lines are for block size of 16 bytes, dashed lines are for block size of 32 bytes. We collected a number of samples for each ORAM configuration equal to a multiple of the Square-Root ORAM shuffle period that is greater than thirty, except in the case of linear scan, for which exactly thirty samples were collected.

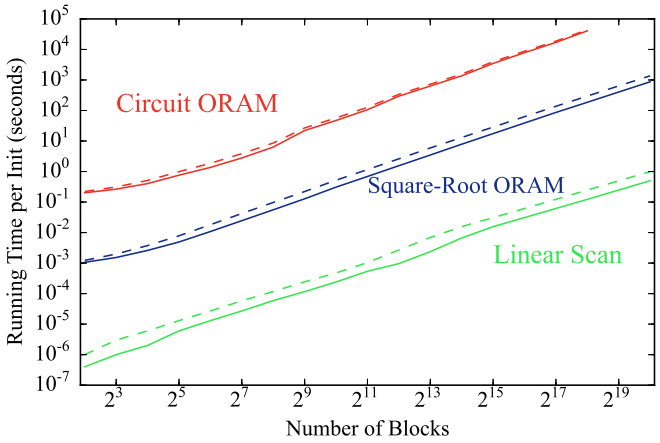


Fig. 10: **Cost of initialization.** Solid lines are for block size of 16 bytes, dashed lines are for block size of 32 bytes.

averages truly are representative, we collected a number of samples for each ORAM configuration equal to a multiple the shuffle period that is greater than thirty, except in the case of linear scan, for which exactly thirty samples were collected.

Breakeven points. Linear scan is preferred to Square-Root ORAM only for very small numbers of blocks. Circuit ORAM is orders of magnitude more expensive for similar parameters, due to its high fixed access cost. Our Square-Root ORAM implementation achieves a very low break-even point with linear scan. When using 4096 or fewer blocks, Circuit ORAM never wins over. And at a block size of 4 bytes, Circuit ORAM remains a suboptimal choice until we have more than 500,000 blocks. But that, in turn, increases initialization cost.

Comparison to Circuit ORAM. In comparing our Square-Root ORAM scheme to Circuit ORAM, we consider initialization and access costs separately since the number of accesses per initialization will vary across applications. Figure 9 shows the per-access wall-clock time for both designs, as well as for linear scan, ignoring initialization.

As expected, Circuit ORAM has the best asymptotic performance, but it also has a very high fixed cost per access, independent of the number of blocks. As a result, Square-Root ORAM performs better than Circuit ORAM for all block counts up to 2^{16}, even ignoring initialization costs. In fact, for block counts less than ~2^{11} linear scan also outperforms Circuit ORAM. These results are consistent with our analysis in Section III-D that Square-Root ORAM has worse asymptotic behavior, but smaller hidden constants.

For any application where the number of accesses is not significantly larger than the number of blocks in the ORAM, initialization cost must be considered. Figure 10 shows the initialization wall-clock times for Square-Root and Circuit ORAM, with parameters matching those in our access-time comparison. For this benchmark, we assume each ORAM must be populated using data already stored in an array of oblivious variables. In such a scenario, a linear scan ORAM requires only that the data be copied; the reported linear scan initialization speed is therefore equivalent to the time required to copy the data.

Initializing Square-Root ORAM is approximately 100 times faster than initializing Circuit ORAM, regardless of block count or block size. The standard way to populate Circuit ORAM is to insert each data element individually, using standard ORAM access operations; thus, the cost scales linearly with the number of blocks to be populated. We hypothesize that most of this speed improvement comes from having fewer network round trips in our initialization process. Circuit ORAM therefore requires $\Theta(N \log N)$ round trips for initialization, while our scheme requires only $\Theta(\log N)$.

C. Oblivious Binary Search

Unlike our other application benchmarks, binary search performs very few accesses relative to the ORAM size. An equivalent search can be performed using a sin-

Benchmark	Parameters	Linear Scan	Square-Root ORAM	Circuit ORAM
Binary Search	1 search	1.00	10.41	3228.69
	2^5 searches	31.87	26.25	3282.40
	2^{10} searches	1019.77	824.81	5040.82
Breadth-First Search	$n = 2^2$	0.09	0.34	4.28
	$n = 2^5$	4.77	4.08	42.66
	$n = 2^{10}$	4569.31	679.63	3750.57
Gale-Shapley	2^3 pairs	-	0.51	6.57
	2^6 pairs	-	145.13	1328.50
	2^9 pairs	-	119405.	188972.
Scrypt	$N = 2^5$	4.11	3.43	34.47
	$N = 2^{10}$	1678.16	293.79	1453.85
	$N = 2^{14}$	*about 7 days*	1919.92	2846.51
	Litecoin	210.92	40.29	247.29

TABLE I: **Summary of benchmark results.** All benchmark results are average measured wall-clock time in seconds for full protocol execution (see individual benchmark sections for details).

gle linear scan, and if only one search is to be performed, the linear scan is always more efficient. Consequently, we varied the number of searches performed for this benchmark, rather than the block size or block count. We benchmarked binary search using a block size of 16 bytes and element counts of 2^{10} and 2^{15}. For arrays of 2^{10} elements, we averaged the running time over 30 samples, and for 2^{15} elements we use 3 samples. A few representative combinations for 2^{15} elements are reported in Table I.

Initialization dominates execution time unless many searches are performed on the same data. As a result, Square-Root ORAM is more than two orders of magnitude better than Circuit ORAM when only one search is performed. For searches of 2^{10} elements, the linear scan method is more efficient than a binary search regardless of the ORAM type or the number of searches performed. Linear scan is initially faster for searches of 2^{15} elements as well, but Square-Root ORAM becomes more efficient than the linear scan method at 2^5 searches. Accesses to a Circuit ORAM of 2^{15} elements are more expensive than accesses to a Square-Root ORAM of the same size, so at this array size, Circuit ORAM will never be more efficient regardless of the number of searches performed.

D. Oblivious Breadth-First Search

Natively-oblivious formulations of Breadth-First Search (BFS) and other graph algorithms have been explored in the past [5]; however, we use a variant of the standard algorithm optimized for use in an oblivious context. It has complexity in $\Theta((V+E)C_{\mathsf{Access}})$, where C_{Access} is the complexity of accessing an element in the underlying ORAM construction. We allow our ORAM implementations to apply arbitrary functions to modify the blocks they access, as opposed to the simple read and write functions shown in Figure 2. This reduces the total number of ORAM accesses by, for example, permitting combined read and update operations. Rather than use an ORAM to house the queue, we use the oblivious queue data structure from Zahur and Evans [39].

We benchmarked our BFS implementation using linear scan, Circuit ORAM, and Square-Root ORAM. We took 30 samples for experiments of n vertices and $\gamma \times n$ edges, with n ranging from 4 to 1024 and γ as 8. For each sample, a fresh set of edges were generated randomly among the chosen number of vertices. A few representative combinations are shown in Table I.

The results of the BFS benchmark roughly follow the pattern established by the microbenchmarks in Section V-B. Small numbers of vertices and edges yield small ORAMs, and linear scan proves to be best in these cases. As the number of vertices or edges begins to rise, Square-Root ORAM quickly becomes more efficient than linear scan. Our BFS implementation uses blocks of only a few bytes each; as a result, Circuit ORAM eventually becomes more efficient than linear scan, but it does not approach the efficiency of Square-root ORAM before the upper bound of our testing range is reached at $n = 2^{10}$. Beyond that point the benchmarks would have required several hours to complete.

E. Oblivious Stable Matching

To explore a benchmark representative of a complex algorithm, we implemented an oblivious version of the Gale-Shapley stable matching algorithm [8]. We followed the textbook algorithm closely. Although we believe there are significant optimizations available in adapting the algorithm for use in MPC, they are beyond the scope of this work.

As a result, our implementation requires $\Theta(n^2)$ accesses of an ORAM with n^2 elements. It also uses of several ORAMs of length n. The most efficient arrangement may be to mix ORAM schemes, but we have not done this. As in our BFS implementation, we used function application to reduce the number of ORAM accesses.

We benchmarked our implementation of Gale-Shapley with both Circuit and Square-Root ORAMs as the underlying structure, but not linear scan since it is clear linear scan cannot be competitive for this benchmark and the expense of executing it on non-trivial sizes would be considerable. The number of pairs to be matched ranged from 4 to 512. When the pair count was less than 128, we collected 30 samples; for pair counts of 128 and 256 we collected 3 samples; for 512 pairs, we collected one sample. Results for few representative configurations are included in Table I.

Square-root ORAM proved more efficient over the entire range we benchmarked, although for sufficiently large sizes Circuit ORAM will eventually do better. For 64 pairs, Square Root ORAM is over 9 times faster (finishing in 145 seconds); for 512 pairs, stable matching requires just over 33 hours using Square-Root ORAM and 52.5 hours with Circuit ORAM.

F. Oblivious Scrypt

To explore the possibility of using ORAMs in a challenging cryptographic application, we implemented the key derivation function *scrypt* [27]. Scrypt was originally intended to be difficult to parallelize, and therefore difficult to break by brute force, even with custom high performance hardware. It achieves this goal by repeatedly enciphering a single block of data, retaining each intermediate result in memory. It then performs a second round of encipherment, mixing the block with an intermediate result from the first round selected according to the current value. In an oblivious context, scrypt *requires* the use of an ORAM of some sort, as the indices of the memory accesses in the second phase depend upon oblivious data generated in the first phase. Due to its unpredictable memory access pattern, the scrypt algorithm is designed to require sequential execution with no significant shortcuts.

With typical parameters, scrypt requires a relatively small ORAM element count. For instance, Litecoin, which uses scrypt as a cryptocurrency proof-of-work, specifies $N = 2^{10}$ elements [22], and Colin Percival, the designer of scrypt, recommends a minimum of $N = 2^{14}$ elements for normal use [27]. On the other hand, Percival recommends that each element be 1KB in size — much larger than required by any of our other application benchmarks. In the course of execution, scrypt performs exactly one access per element.

We tested scrypt using the recommended parameters and test vectors from the scrypt specification [27], $r = 8$ and $p = 1$, and we varied N from 4 to 2^{14}. In addition, we benchmarked the parameters used by Litecoin, ($r = 1$, $p = 1$, $N = 2^{10}$). A few representative combinations are presented in Table I. As in the other benchmarks, linear scan is marginally more efficient when the number of blocks (N) is small. Otherwise, Square-Root ORAM is preferred; it exceeds the performance of linear scan by approximately one order of magnitude when $N = 2^{10}$, and this ratio improves as N increases.

The largest parameters we benchmarked are Percival's recommended minimum parameters ($r = 8$, $p = 1$, $N = 2^{14}$), which he originally chose on the basis that they required less than 100ms to execute on contemporary hardware, this being what he considered a reasonable threshold for interactive use [27]. On our EC2 test node, the reference (non-oblivious) scrypt implementation requires 35ms with the same parameters. With Square-Root ORAM as the underlying primitive, execution took 32 minutes, compared with 47 minutes for Circuit ORAM. The large block size required by scrypt causes block access time to form a greater portion of the total cost than in our other application benchmarks. As a result, Circuit ORAM becomes competitive earlier than in the other cases. We did not benchmark linear scan for the recommended parameters; we estimated that it would require roughly 7 days to complete, well beyond what could reasonably be considered useful in practice.

Even with Square-root ORAM, scrypt requires 55,000 times longer to execute with real-world parameters as an MPC protocol than it does to execute conventionally. This is almost certainly too expensive to be practical for any interactive application today, but shows that even complex algorithms designed intentionally to be expensive to execute are not beyond the capabilities of general-purpose MPC today.

VI. Conclusion

The success of MPC depends upon enabling developers to create efficient privacy-preserving applications, without requiring excessive effort, expertise, or resources. It is important that MPC protocols be compatible with conventional programming techniques and data structures with depend on random access memory. Our Square-Root ORAM design provides a general-purpose oblivious memory that can be used anywhere a programmer would normally use an array. We have presented a new approach for designing ORAMs for MPC, which demonstrates how hierarchical ORAM designs can be implemented efficiently, and how they can overcome the high initialization costs and parameter restrictions of previous ORAM designs. This represents a step towards a programing model for MPC in which standard algorithms can be efficiently implemented as MPCs, using oblivious memory just like conventional memory is used today.

Acknowledgments

We would like to thank Yilei Chen and Oxana Poburinnaya for engaging discussions during the early phases of this work. The Gale-Shapley benchmark was suggested by abhi shelat.

This work was partially supported by grants from the National Science Foundation SaTC program (Xiao Wang and Jonathan Katz supported in part by NSF Award CNS-1111599; Jack Doerner, David Evans, and Samee Zahur supported in part by NSF Award CNS-1111781), the Air Force Office of Scientific Research, and Google. Work of Mariana Raykova, Samee Zahur and Xiao Wang was done in part while at SRI International and was supported by NSF awards CNS-1421102,1633282 and CCF-1423296. Work of Adrià Gascón was supported by the SOCIAM project, funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/J017728/2, and the NSF award CCF-1423296.

References

[1] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to Efficiently Evaluate RAM Programs with Malicious Security. In *EUROCRYPT*, 2015.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *ACM Conference on Computer and Communications Security*, 2013.

[3] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE Symposium on Security and Privacy*, 2013.

[4] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward. In *ACM Conference on Computer and Communications Security*, 2015.

[5] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ACM Symposium on Information, Computer and Communications Security*, 2013.

[6] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally Secure Constant-rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation. In *Theory of Cryptography*, 2006.

[7] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-Party ORAM for Secure Computation. In *ASIACRYPT*, 2015.

[8] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[9] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using it Efficiently for Secure Computation. In *Privacy Enhancing Technologies*, 2013.

[10] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private Database Access with HE-over-ORAM Architecture. In *Applied Cryptography and Network Security*, 2015.

[11] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM Symposium on the Theory of Computing*, 1987.

[12] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3), 1996.

[13] Michael T. Goodrich and Michael Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *International Colloquium on Automata, Languages and Programming*, 2011.

[14] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation. In *ACM-SIAM Symposium on Discrete Algorithms*, 2012.

[15] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure Two-Party Computation in Sublinear (Amortized) Time. In *ACM Conference on Computer and Communications Security*, 2012.

[16] Yan Huang, David Evans, and Jonathan Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *Network and Distributed Systems Security Symposium*, 2012.

[17] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation using Garbled Circuits. In *USENIX Security Symposium*, 2011.

[18] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. In *ASIACRYPT*, 2014.

[19] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security*, pages 1–20. Springer, 2009.

[20] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. *Automata, Languages and Programming*, 2008.

[21] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (In)security of Hash-Based Oblivious RAM and a New Bal-

ancing Scheme. In *ACM-SIAM Symposium on Distributed Algorithms*, 2012.

[22] Litecoin Project. scrypt.cpp. https://github.com/litecoin-project/litecoin/blob/master-0.10/src/crypto/scrypt.cpp, 2015.

[23] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating Efficient RAM-Model Secure Computation. In *IEEE Symposium on Security and Privacy*, 2014.

[24] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *IEEE Symposium on Security and Privacy*, 2015.

[25] Steve Lu and Rafail Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. In *Theory of Cryptography Conference*, 2013.

[26] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a Secure Two-Party Computation System. In *USENIX Security Symposium*, 2004.

[27] Colin Percival. Stronger key derivation via sequential memory-hard functions. http://www.tarsnap.com/scrypt/scrypt.pdf, 2009.

[28] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In *CRYPTO 2010*, 2010.

[29] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, 2009.

[30] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT 2011*, 2011.

[31] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE Symposium on Security and Privacy*, 2015.

[32] 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 *ACM Conference on Computer and Communications Security*, 2013.

[33] Abraham Waksman. A Permutation Network. *Journal of the ACM*, 15(1), January 1968.

[34] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM Conference on Computer and Communications Security*, 2015.

[35] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure Computation of MIPS Machine Code. Cryptology ePrint Archive, Report 2015/547, 2015. http://eprint.iacr.org/2015/547.

[36] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *ACM Conference on Computer and Communications Security*, 2014.

[37] Peter Williams and Radu Sion. Single Round Access Privacy on Outsourced Storage. In *ACM Conference on Computer and Communications Security*, 2012.

[38] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets (Extended Abstract). In *IEEE Symposium on Foundations of Computer Science*, 1986.

[39] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *IEEE Symposium on Security and Privacy*, 2013.

[40] Samee Zahur and David Evans. Obliv-C: A Lightweight Compiler for Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153, 2015. http://oblivc.org.

[41] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT*, 2015.

APPENDIX

Figure 11 shows the actual Obliv-C source code of our ORAM construction, copied verbatim. The **obliv** keyword denotes secret variables. The variable ram→cpy is a structure with block size and copy constructor information. Since block size is only known at runtime, a pointer to array[i] must be obtained by calling element(ram→cpy, array, i). The actual Obliv-C code closely follows the pseudocode presented in Figure 5.

```
static void* element(OcCopy* cpy, void* arr, int i) obliv
  { return i * cpy→eltsize + (char*)arr; }

void ocSqrtOramAccess(OcSqrtOram* ram, obliv int index,
                      ocBlockAccessFunction fn, void* data)
{
  int i;
  obliv bool foundi = false;
  // Scan through stash
  for (i=0; i<ram→time; ++i) obliv if (index == ram→stashi[i])
  { fn(ram→cpy, element(ram→cpy, ram→stash, i), data);
    found=true;
  }

  // Fake/unfake posmap lookup
  int lookupIndex = ram→pos→getPos(ram→pos, index, found);

  // Access one more element from shuffled array
  ocCopy(ram→cpy, element(ram→cpy, ram→stash, ram→time),
                  element(ram→cpy, ram→shuff, lookupIndex));
  ram→usedShuff[lookupIndex] = true;
  ram→stashi[ram→time] = ram→shuffi[lookupIndex];
  obliv if(!found)
    fn(ram→cpy, element(ram→cpy, ram→stash, ram→time), data);
  ram→time++;
  if (ram→time == ram→period) {
    ocSqrtOramRefresh(ram);
  }
}
```

Fig. 11: Obliv-C implementation of the Access function of our ORAM construction