SCORAM: Oblivious RAM for Secure Computation*

Xiao Shaun Wang
University of Maryland
wangxiao@cs.umd.edu

Yan Huang
Indiana University
Bloomington
yh33@indiana.edu

T-H. Hubert Chan
University of Hong Kong
hubert@cs.hku.hk

abhi shelat
University of Virginia
abhi@virginia.edu

Elaine Shi
University of Maryland
elaine@cs.umd.edu

ABSTRACT

Oblivious RAMs (ORAMs) have traditionally been measured by their *bandwidth overhead* and *client storage*. We observe that when using ORAMs to build secure computation protocols for RAM programs, the *size* of the ORAM circuits is more relevant to the performance.

We therefore embark on a study of the *circuit-complexity* of several recently proposed ORAM constructions. Our careful implementation and experiments show that asymptotic analysis is not indicative of the true performance of ORAM in secure computation protocols with practical data sizes.

We then present SCORAM, a heuristic *compact* ORAM design optimized for secure computation protocols. Our new design is almost 10x smaller in circuit size and also faster than all other designs we have tested for realistic settings (i.e., memory sizes between 4MB and 2GB, constrained by 2^{-80} failure probability). SCORAM makes it feasible to perform secure computations on gigabyte-sized data sets.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General-security and protection

Keywords

Oblivious RAM; Secure Computation

1. INTRODUCTION

Secure two-party computation allows Alice, who holds private input x and Bob, who holds private input y, to jointly compute $f(x, y)$ without revealing any information other than the output $f(x, y)$. All known efficient constructions of such generic cryptographic protocols require an *oblivious* representation of the function f to ensure that the control flow of the algorithm does not depend on its input and therefore leak partial information. The standard approach to creating an oblivious representation is to generate a boolean circuit from the description of f. This strategy is employed by dozens of prior works [1, 14, 16, 17, 19, 22, 23, 29] on secure computation.

When f is given as a RAM (Random Access Memory model) program, transforming f into a binary circuit may be problematic. A naive transformation replaces each indexed access to memory with a scan of the entire memory in order to keep the index hidden. To overcome this issue, Gordon *et al.* [12] used an Oblivious RAM (ORAM) data structure proposed first by Goldreich [7] to compile RAM programs into secure computation protocols.

Intuitively, ORAM is a technique to transform a memory access (with *secret* index i) into a sequence of memory accesses (whose indices are revealed to the adversary but appear independent of the secret value i). ORAM techniques have been widely studied in other contexts [2–4, 7, 9–11, 18, 26–28, 30, 34, 36–38]. However, the goals of these prior works were (1) reducing the bandwidth overhead between the client and server; (2) reducing the *client storage*; and (3) reducing the server's overall memory overhead. Remarkably, state of the art approaches to ORAM design limit the overhead in all three aspects to various combinations of $O(\log^c(n))$ where $c \in \{0, 1, 2, 3\}$.

Towards large-scale secure computation. Secure computation is presently limited to small instances because large datasets incur very-large overheads in the circuit-model. The RAM model offers an asymptotically more efficient approach to many types of secure computation so as to potentially scale to large datasets. Indeed, the development of practical ORAM techniques and demonstrations of practical ORAM implementations [34] have lead to notable works that demonstrate how ORAM-based secure computation protocols enable dramatic efficiency improvements in processing large, secret datasets. For example, Gordon *et al.* [12] show how repetitive binary searches can be securely computed in amortized sublinear time. Liu *et al.* [19] show how, under big data sizes, RAM-model secure computation significantly outperforms the circuit model even for *run-once* tasks, including KMP-string matching and shortest path. Keller and Scholl [] also show how the PATH-ORAM scheme can lead

*This research is partially funded by the National Science Foundation under grants CNS-1314857 and 0845811, a Sloan Research Fellowship, Google Faculty Research Awards, a grant from Amazon AWS, a grant from Hong Kong RGC under the contract HKU719312E, as well as DARPA and AFRL under contracts FA8750-11-C-0080 and FA8750-14-C-0057. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the funding agencies.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660365 .

ORAM	Circuit Size (Asymptotic Bounds)	Circuit Size (gates) $N = 2^{20}$	Circuit Size (gates) $N = 2^{29}$	Number of Inputs $N = 2^{20}$	Number of Inputs $N = 2^{29}$
Linear Scan ORAM	$O(DN)$	142.6M	≈82678.1M	33.5M	17180.0M
LO ORAM	$\mathbf{O((D + C_{PRF})logN)}$				
CLP ORAM (w/ oblivious queue)	$O(\log^4 N + D \log^2 N)\omega(1)$				
CLP ORAM (w/o oblivious queue)	$O(\log^5 N + D \log^3 N)\omega(1)$	29.4M	121.8M	0.6M	2.1M
Binary Tree ORAM	$O(\log^4 N + D \log^2 N)\omega(1)$	38.5M	127.7M	7.0M	24.2M
Naive Path ORAM	$O(\log^4 N + D \log^2 N)\omega(1)$	56.2M	163.1M	**0.1M**	**0.3M**
Path-SC ORAM	$\tilde{O}(\log^3 N + D \log N)\omega(1)$	37.2M	111.7M	**0.1M**	**0.3M**
SCORAM	N/A (heuristic)	**4.6M**	**13.0M**	0.3M	0.9M

Table 1: **Performance metrics for different ORAM schemes for secure computation.** N is the number of blocks in the ORAM, D is number of bits in each block (i.e., the payload bit length), C_{PRF} is the circuit complexity of a PRF function, and the security parameter is set to be $O(\log N)\omega(1)$. We use the notation $g(N) = O(f(N))\omega(1)$ to denote that for any $\alpha(N) = \omega(1)$, it holds that $g(N) = O(f(N)\alpha(N))$. The notation $\tilde{O}$ hides $\log \log N$ factors. All concrete measurements are for $D = 32$ bits with failure probability set to 2^{-80} and include all levels of recursion for recursive ORAM constructions. The circuit size reports all gates, and the number of inputs defines cost of input preprocessing, e.g., OT. For CLP ORAM, a $O(\log N)$ oblivious queue [24] can lower the asymptotic bound but will introduce a larger circuit size in practice than trivial oblivious queue because of the small queue size. Details can be found in Section 2.2.

to dramatic improvements for certain secure computation problems. This work tackles the problem of finding very efficient ORAM constructions to enable secure computation on giga-byte sized datasets.

1.1 Contribution

We embark on a comprehensive study of practical secure computation ORAM techniques. We do so via both theoretical analysis of several metrics used to judge ORAMs and experiments performed on optimized implementations of 4 state-of-the-art ORAM schemes. We report a new heuristic ORAM design that outperforms all other ORAMs we considered.

Our first observation is that traditional measures of ORAM schemes do not properly indicate the ORAM performance in secure computation setting. Previously, ORAM was primarily considered in storage outsourcing [9, 32, 33], and secure processor execution [5, 21] settings. Thus, ORAM constructions were mainly evaluated by the *bandwidth overhead* (i.e., the number of data blocks retrieved per memory query).

Since the client-side computational logic needs to process secret values (e.g., the original index), their cost cannot be ignored as in traditional ORAMs designed merely for outsourcing storage. On the contrary, the overhead due to *securely* computing the client-side ORAM logic can easily dominate the overall cost in both bandwidth and CPU cycles. Therefore, the *circuit complexity* of the ORAM algorithm plays a critical role in evaluating the efficiency of ORAM schemes used in secure computation.

Re-evaluate and improve existing ORAMs. We conduct a systematic evaluation of several state-of-the-art ORAM schemes (e.g., Binary Tree ORAM [30], Path ORAM [34], CLP ORAM [3]) in the secure computation setting. Table 1 reports both the theoretical and concrete complexity of these existing ORAM schemes.

We find that, asymptotically speaking, the Binary Tree ORAM by Shi *et al.* [30] performs the same as a naively implemented Path ORAM [34], although Path ORAM is asymptotically faster in the data outsourcing scenario. In fact, due to the circuit size, Path ORAM is significantly slower for practical parameter settings. Next, we derive Path-SC ORAM, an optimized construction of Path ORAM that is $O(\log n)$ faster than its naive implementation. However, because of the 3 oblivious sorts (resulting in a large constant factor in practice), its performance in practical parameter settings is still inferior to those theoretically slower schemes such as Binary Tree ORAM [30] and CLP ORAM [3].

A new ORAM scheme. While we do not present an *asymptotically* more efficient ORAM, we propose a scheme that is *empirically* the most efficient. We argue that within $\mathsf{poly} \log N$ complexity ranges, optimizing for asymptotic performance can be misguided. For conceivable data sizes ($2^{20} \sim 2^{40}$ blocks), $\log N$ is typically 20 to 40, and can be easily dominated by even moderate constants (e.g. 100). Similar observations of asymptotics vs. practical performance are not uncommon, e.g. by Stefanov *et al.* in constructing small-domain PRPs [31].

Our new ORAM scheme, SCORAM, is almost **10x** smaller in circuit size than the best previous one derived from a tranditional ORAM protocol. It is significantly faster than all existing ORAM constructions in secure computation setting. Further, this new scheme can also be used to build efficient oblivious data structures [35] for secure computation. Our implementation of SCORAM will be available online at `http://www.oblivm.com`.

1.2 Related Work

Gordon *et al.* [12] studied the feasibility of constructing sublinear-time secure computation protocols. They showed generically how any function $f(\cdot, \cdot)$ that can be computed in time t and space s in the RAM model can be securely computed by a protocol that requires amortized time $O(t \cdot \mathrm{poly} \log(s))$. They reported an implementation of Binary Tree ORAM; using security parameter less than 20, and $N = 2^{20}$, their ORAM scheme requires roughly 6.6M non-free gates and 12M total gates per memory access. To the best of our ability, we attempted to recreate their parameters: our implementation of the same scheme required 3.3M non-free gates and 16M total gates; while our best scheme

configured at a much higher security parameter results a circuit approximately 1/3 of theirs . See Section 6 for details.

Gentry et al. [6] optimized the binary tree ORAM for secure computation, reducing the tree height using larger buckets. We do not include the scheme in Table 1 because their scheme is subsumed both asymptotically and empirically by Path ORAM [34]. This observed can be verified with a simple back-of-the-envelope calculation: Their eviction algorithm is similar to Path ORAM's, except that they directly compute where a block should be dropped along the path — closest to the leaf respecting invariant. Naively implementing their eviction would result in $O(A^2)$ overhead where A is the total number of blocks on the path, i.e., A = (bucket size)*(path length). Notice that Path ORAM also has $O(A^2)$ overhead for a naive eviction circuit, but Path ORAM's A value is both asymptotically and empirically smaller than that of Gentry et al. In both schemes, we can have an asymptotically smaller eviction circuit with oblivious sorting, but as we show, oblivious sorting introduces a large constant such that the practical performance is worse than that of the original binary-tree ORAM.

Keller and Scholl [15] implemented secure oblivious data structures using both the Binary Tree ORAM [30] and a variation of Path ORAM [34]. However, from their result, the variation does not perform better than Binary Tree ORAM even with small security parameters. In contrast, SCORAM outperforms Binary Tree ORAM by 7x even with large security parameter(80).

Lu and Ostrovksy [20] proposed asymptotically the best ORAM in the literature. Their 2-server design can perform a sequence of n reads or writes with $O(\log n)$ *amortized* overhead per access while using $O(n)$ storage for the servers and $O(1)$ client memory. As these parameters meet the lower-bound for the performance of a single-server ORAM, it appears to be a perfect candidate for ORAM in secure computation. Unfortunately, there are two serious practical bottlenecks to the implementation of LO ORAM.

To understand the first issue, recall that the LO ORAM was built upon the KLO ORAM [18] which was further built upon[1] the map-reduce based cuckoo-hashing ORAM of Goodrich and Mitzenmacher [9]. In order to read or write at index x, the scheme iteratively queries a hierarchy of hash tables $H_k, H_{k+1}, \ldots, H_L$ with either x or a dummy address t depending on whether x has been found or not. The security relies on issuing the dummy query once x has been found in order to maintain the invariant that every lookup is unique (i.e, there is never a lookup for the same address x at any two levels in the hierarchy). This means that read queries require *several sequential* executions of separate secure computation protocols; in contrast, tree-based ORAM designs require only 1 secure computation protocol to be run per read/write operation.

The second serious problem is a large overhead constant for cuckoo hashing. All cuckoo-hashing based ORAMs depend on a lemma proven by Goodrich and Mitzenmacher [9] which bounds the collision rate of a cuckoo-hashing structure via a combinatorial analysis of a graph G that is based on the cuckoo-hashing function. This lemma requires the cuckoo hash table size to be at least $\Omega(\log^7(N))$; this technique therefore only starts to beat the Linear Scan ORAM when $N > 2^{37}$.

[1]KLO pointed out a subtle security issue that affects almost all cuckoo-basing based ORAM schemes, and explain how to fix it.

2. BACKGROUND: TREE-BASED ORAMS

Notation. We use N to denote the number of (real) data blocks in ORAM, D to denote the bit-length of a block in ORAM, Z to denote the capacity of each bucket in the ORAM tree, and λ to denote the ORAM's statistical security parameter. When discussing binary trees of depth L in this paper, we say the *leaves* are at level 0 and the root is at level $L-1$. Although unconventional, this simplifies the description of our algorithms.

2.1 Tree-based ORAM Construction

Shi *et al.* [30] proposed a new binary-tree based framework for constructing a class of ORAM schemes. Many recent efficient ORAM schemes [3,30,34] extend this construction, so we briefly review this framework below. The key difference between the schemes is the choice of *eviction* strategy.

Data organization. The server organizes N *blocks* into a binary tree of height $L = \log N$; each node of the tree is a *bucket* containing Z blocks of the form:

$$\{\texttt{idx}||\texttt{label}||\texttt{data}\},$$

where `idx` is the index of a block, e.g., the (logical) address of desired block; `label` is a leaf identifier specifying the path on which the block resides; and `data` is the payload of the block.

The client stores a *position map*, mapping memory addresses to *leaf labels*. Position map storage can be reduced to $O(1)$ by recursively storing the position map in a smaller ORAM (see [30] for details). These leaf labels are assigned randomly and are reassigned as blocks are accessed. If we label the leaves from 0 to $N-1$, then each label is associated with a path from the root to the corresponding leaf. Tree-based ORAMs maintain the *invariant* that a block marked label resides either on the path leading to the corresponding leaf node specified by label or in the stash.

Operations. Tree-based ORAM support three operations. Among these, the Eviction algorithm is the key difference between schemes.

- ReadAndRemove: Given an index `idx`, the client looks up its label from the position map, and fetches all blocks on the path leading to label. The client finds the block `idx` (due to the main invariant) and removes it from the path.
- Add: The retrieved block is potentially updated, reencrypted and written back.
- Eviction: Percolate blocks towards leaves such that no bucket will overflow except with negligible probability. Various ORAMs use different eviction schemes which we explain below.

Recursion. Instead of storing the entire position map in the client's local memory, the client can store it in a smaller ORAM on the server. In particular, this position map ORAM needs to store $N \log(N)$-bit labels. By storing χ labels in one block, this ORAM only needs N/χ blocks. Finally, by applying recursion, the position map can be reduced to $O(1)$ size, after which a linear scan ORAM can be used. See Section 5 for a discussion of how we set the recursion parameters. Unless otherwise noted, our discussion of the complexity of the eviction strategy applies to a single ORAM (i.e., not taking into consideration the recursion, which applies equally to all tree-based strategies).

2.2 Various Eviction Strategies

Original binary-tree ORAM. The original Binary Tree ORAM scheme [30] adopts random eviction: with every data access, pick two random buckets from each level, and evict one block from each selected bucket by placing the block into the correct child node (subject to the invariant). Shi *et al.* show that this eviction strategy requires a bucket size of $O(\log N)\omega(1)$ to avoid overflow except with negligible probability.

Path ORAM. Path ORAM adopts a greedy eviction strategy that works with a *stash* maintained in client storage. Blocks on the path P are first unioned with the stash. Each block in this set is then placed as close as possible to its destination leaf in path P subject to the path invariant. Stefanov *et al.* [34] show that this aggressive eviction strategy works with a bucket size of 4.

CLP ORAM. In CLP ORAM [3], internal nodes of the binary tree contain $O(\log \log N)$ blocks per bucket, and the stash is replaced by a queue of length $\omega(\log^2 N)$ that requires add, pop and find operations. The eviction strategy works as follows: in each level pick the "deepest" block on the eviction path and push it to the next level. If blocks in a bucket trigger a certain property, a special overflow procedure will be called, which will add a block in the current bucket to the queue. However, the fact that every level can overflow makes the worst case running time much worse then average case: the overflow procedure needs to be called $\log N$ times in the worst case. An asymptotically better oblivious queue [24] can be used for CLP ORAM, however, linear scan performs better in practice because the size of queue is small.

2.3 ORAM for Secure Computation

Unlike the traditional ORAM scenario, in the Ostrovsky-Shoup framework, both the server memory and the client storage of the ORAM are secret-shared between the parties[2]. Each *instruction* of a RAM program consists of an address to read from memory, an operation to perform, and an address to write back to memory. These three steps are accomplished via a secure computation protocol between the parties that takes as input (a) secret shares of the memory, (b) secret shares of the ORAM client state, (c) and secret shares of the program state.

In the RAM-model secure computation framework, both the program logic and memory operations are implemented over secure computation. The cost of memory operations can be significant since evaluating ORAM over secure computation is expensive. For tree-based ORAM schemes, typically, the eviction procedure can account for the majority of the cost. In the next section, we start with implementing Naive Path ORAM in circuit.

3. PATH-SC ORAM

We first investigate Path ORAM since it has the smallest empirical bandwidth overhead among all tree-based ORAMs. Naturally a good question is whether we can implement Path ORAM with a small circuit as well. Since ReadAndRemove and Add algorithm are trivial to turn into $O(D \log N)\omega(1)$-sized circuits, we focus our discussion on how to implement Path ORAM's eviction algorithm in circuit.

[2]We note that Gordon *et al.* propose an asymmetric division of the server and client state to enable one party in the secure computation protocol to have a sublinear number of input bits.

```
1: P[0..L − 1][Z] stores the block to be put back
2: for i from 0 to L − 1 do //from leaf to root
3:     for j from 1 to Z do
4:         for k from 1 to LZ + stsize do
5:             if LCA(A[k].label, P) ≤ i then
6:                 P[i][j] := A[k]
7:                 A[k] := ⊥
```

Figure 1: **Naive oblivious algorithm for Path ORAM's Eviction.** Variable A denotes a buffer created by concatenating the stash and the path P read in Path ORAM. This algorithm writes as many blocks back to P as possible, packing them as close to the leaf as possible.

Expressing circuits. In the remainder of the paper, we will need to describe circuit constructions. Since oblivious algorithms with constant client memory size can be easily transformed into circuits preserving complexity, we henceforth equate the notions of *oblivious algorithms* and *circuits*, and describe circuits using oblivious algorithms.

Notations. Suppose each of p_1 and p_2 represents a leaf or a root-to-leaf path. We use $\mathsf{LCA}(p_1, p_2)$ to denote the level of the lowest common ancestor of the leaves, or equivalently the node where the two paths diverge when traversing from the root. In the remainder of the paper, we sometimes *use the notation* label *and a path* p *interchangeably*, since a path p is defined by the label of a leaf node. When we sort lexicographically according to a key pair $(\mathsf{key}_1, \mathsf{key}_2)$, we mean first sort according to key_1, and if there is a tie on key_1, then we sort according to key_2.

Let A denote a buffer created by concatenating the stash with a path p (the data read path) in Path ORAM. Path ORAM's eviction writes as many blocks from A to p as possible and packs them as close to the leaf as possible.

Naive $O(D \log^2 N)$ eviction circuit. A naive way to turn Path ORAM's eviction algorithm into a circuit would result in a $O(D \log^2 N)\omega(1)$-sized circuit. We describe this naive method in Figure 1.

3.1 A New $\tilde{O}(D \log N)$ Eviction Circuit

The rearrangement problem. Path ORAM's eviction can be recast as a rearrangement problem: we would like to obliviously rearrange the entries in A (by pairwise swapping) with respect to the following conditions:

1. The blocks in $A[1...LZ]$ will be written back to the path P, where the block at $i = kZ+j$ (where $k = \lfloor \frac{i-1}{Z} \rfloor$ and $1 \le j \le Z$) will go to bucket k. We require that for each i, if $A[i]$ is a real block, then $\mathsf{LCA}(A[i].\mathsf{label}, P) \le k$. If less than Z real blocks are assigned to a bucket, dummy blocks will be added to that bucket.
2. For the real blocks that cannot be written in the path, they will be stored in $A[LZ+1...LZ+\mathsf{stsize}]$. Hence, if $A[LZ + \mathsf{stsize} + 1]$ contains a real block, this indicates stash overflow.

Circuit construction. First, we add three extra fields bucket, offset and dummy (used only in this improved eviction algorithm) to each entry $A[i]$. bucket takes values from $\{0, \ldots, L-1\} \cup \{\perp\}$. For a dummy block, bucket is set to $\perp$. For a real block, $\mathsf{bucket} := \mathsf{LCA}(\mathsf{label}, P)$, where label denotes the leaf label of the block. In other words, bucket

1. **Initialization.** For each $i \in \{1, 2, \ldots, LZ\}$: let$A[i]$.bucket := $\mathsf{LCA}(A[i].\mathsf{label}, P)$, let $A[i]$.dummy denote whether $A[i]$ is a dummy block.
2. **O-sort: real before dummy.** Oblivious sort based on the key bucket. After sorting, all real blocks come first, in bucket ascending order.
3. **Scan A to compute the final offset (from leaf) for each block.** We give the algorithm to calculate offset from bucket in Figure 4.
4. **Append dummy.** Append LZ dummy blocks at the end whose offset are $1, \ldots, LZ$, respectively.
5. **O-sort: reorder based on offset.** Oblivious sort A by offset (where blocks with offset $= \perp$ are put at the end).
6. **Suppress unnecessary dummy.** Scan A to "eliminate" *unnecessary* dummy blocks. A dummy block is considered *unnecessary* if it is preceded by a real block with the same offset value. We "eliminate" this dummy block by setting its offset to "$\perp$".
7. **O-sort: fall into place.** Sort A by offset so that unnecessary dummys are moved to the end A.

Figure 2: **Eviction algorithm of** Path-SC ORAM.

is the lowest level (i.e., closest to leaf) where the block is allowed to reside on the path P. Recall that we assume bucket $= 0$ refers to the leaf level while bucket $= L-1$ refers to the root level. offset takes values from $\{1, \ldots, LZ\} \cup \{\perp\}$. dummy indicates if a block is dummy.

The improved eviction algorithm is described in Figure 2. At the end of the algorithm, the first Z blocks of A can go to the leaf, path, and the next Z blocks should go to leaf but one level, and so forth. The idea of using oblivious sorting to design efficient oblivious algorithms is also used in other scenarios [25].

In Appendix A, we also consider how to construct a low-depth circuit for our new eviction algorithm.

Examples. The most sophisticated part of the algorithm is Step 3 in Figure 2, i.e., computing the offset of blocks. This part of the algorithm is further explained in Figure 4. Below we give an example of this step. Assume that $Z = 3$. Then, a sequence of bucket values [0 0 0 0 1] should result in offset values [1 2 3 4 5]. Additionally, bucket values [0 1 1 1 1] should result in offset values [1 4 5 6 7].

An example of the full eviction circuit is given in Figure 3.

```
1: available := 1
2: for i from 1 to LZ + stsize do
3:     if available ≥ Z * A[i].bucket + 1 then
4:         A[i].offset := available
5:         available := available + 1
6:     else
7:         A[i].offset := Z * A[i].bucket + 1
8:         available := A[i].offset + 1
```

Figure 4: **Computing** offset **from** bucket.

THEOREM 1. *The Path-SC ORAM with* Eviction *described above can be fully implemented (i.e., including all the recursive ORAMs used to store the position map) in* $\tilde{O}(\log^3 N + D \log N)$ *boolean gates.*

PROOF. The circuit to implement the data level ORAM requires $O(D \log N \log\log N)$ gates due to the oblivious sorting operations in Figure 2. For recursion levels, we use $O(\log N)$ for each block, and with $O(\log N)$ levels of recursion, the total circuit size for position map recursion levels is $O(\log^3 N \log\log N)$. Therefore, the total circuit size for all levels is $\tilde{O}(\log^3 N + D \log N)$. □

Findings. We implement our Path-SC ORAM and compare its empirical performance with that of the Binary Tree ORAM and CLP ORAM. While Path-SC ORAM is asymptotically superior to both Binary Tree ORAM and CLP ORAM in terms of circuit size, for practical ranges of N, empirical results suggest that both the Binary Tree ORAM and CLP ORAM perform better. Detailed empirical results are presented in Section 6.

Upon closer examination, we realize that Path-SC ORAM requires 3 oblivious sorts in its circuit construction, which leads to a very large constant. This motivated our search for a scheme that could avoid sorts altogether.

4. A HEURISTIC ORAM

We devise a heuristic ORAM scheme called SCORAM optimized for empirical performance (as opposed to asymptotical performance). Our key idea is to have an effective eviction algorithm that can be implemented in small circuit. In this section, we focus on presentation of the algorithm, leaving optimal parameter choices to the next section.

4.1 SCORAM

Our SCORAM is another tree-based ORAM with a novel Eviction algorithm. Overall, Eviction will perform `flush()` (Algorithm 1) α times. In our implementation, we choose $\alpha = 4$ which we determine to be the optimal choice (see Section 5 for more details). Below we explain the intuition.

Greedy push pass (lines 7 to 10) We opt for a greedy push pass similar to that of the CLP ORAM, but avoid their *bucket overflows* handling strategy. First, a random path is selected for eviction with every data access. Next, for each bucket from root to the leaf-1 level on the selected path: pick a block that can be evicted deepest along this path, and push it to the child bucket if there is room.

In CLP ORAM, a bucket overflow occurs when more than half of the bucket capacity number of blocks can be evicted to one of its children. Overflow events are indicative of getting too crowded in some parts of the tree. Chung et al. handle this event by choosing a block to remove from the bucket, remapping its label, and putting it back into the queue. This is an expensive operation, since (1) removing (adding) blocks from a bucket (to the stash) requires a linear scan; (2) it needs to be done for every bucket on the eviction paths because we cannot reveal where the actual overflows happen; (3) updating the (recursively stored) position map is also very expensive. Although some of the above

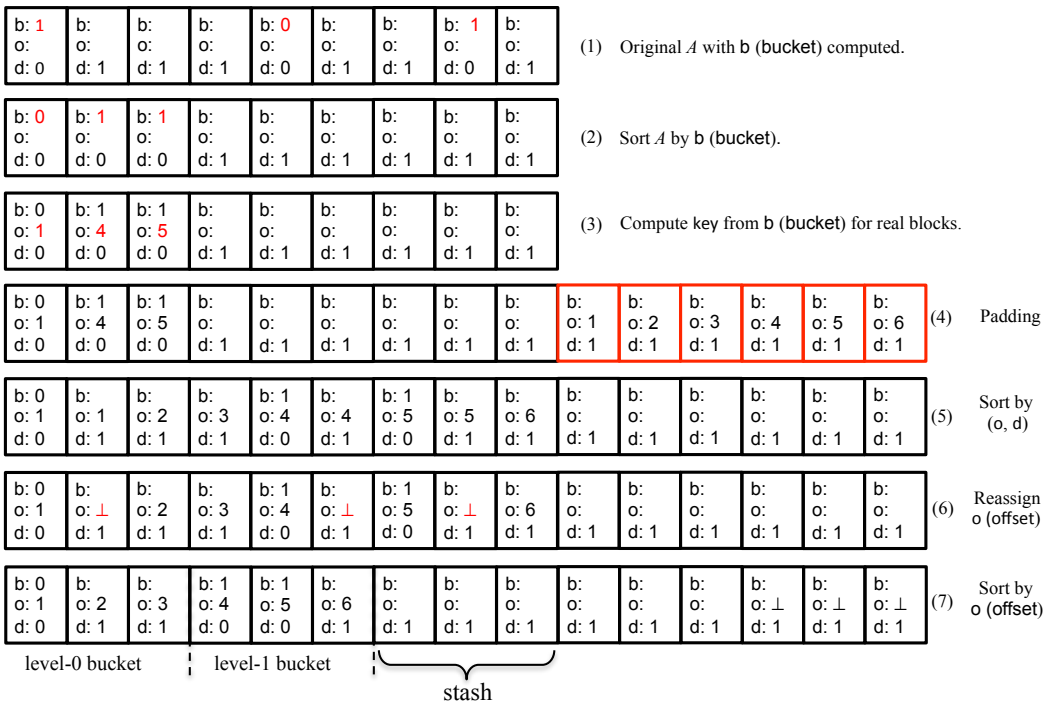


Figure 3: **A toy example of** Path-SC ORAM **eviction algorithm.** Here we assume $L = 2, Z = 3$ and stsize $= 3$. b stands for the bucket field while k the offset field. d=1 indicates dummy blocks, where the contents of other fields are not applicable.

Algorithm 1 `flush()`

1: `path` := UniformRandom$(0, ..., N-1)$
2: `bucket`$[0, ..., L-1]$:= array of buckets from leaf to root
3: B_1 := the block in the stash with smallest LCA(`path`, B.label).
4: **for** i from 0 to $L-1$ **do** *(from leaf to root)*
5: **if** `bucket`$[i]$ is not full and LCA(`path`, B.label) $\leq i$ and B_1 has not been added already **then**
6: Add B_1 to `bucket`$[i]$.
7: **for** i from $L-1$ to 1 **do** *(from root to leaf)*
8: B_2 := the block in `bucket`$[i]$ with smallest LCA(`path`, B_2.label).
9: **if** `bucket`$[i-1]$ is not full and LCA(`path`, B_2.label) $< i$ **then**
10: Move B_2 from `bucket`$[i]$ to `bucket`$[i-1]$.

overflow handling logic can be implemented asymptotically better using oblivious sorting, this actually worsens the empirical overhead due to the large constant factor associated with oblivious sorting.

In contrast, in SCORAM, a block will only be evicted if the child bucket is not full. Unfortunately, such a "greedy push pass" alone would make the eviction less effective, as it is more likely for a bucket to be full and the stash could grow unduly large. This motivates our idea of compensating with a "reverse dropping pass".

Reverse dropping pass (lines 3 to 6). Pick a block from stash that can be pushed deepest along the path, then put it into the bucket as deep along the path as possible. This can be implemented by scanning the eviction path in reverse order from leaf to root and placing the block into the first non-full bucket that satisfies the block's path-invariant.

Security. Security of the scheme follows as per all other tree-based ORAMs; the server's view observes accesses along random paths.

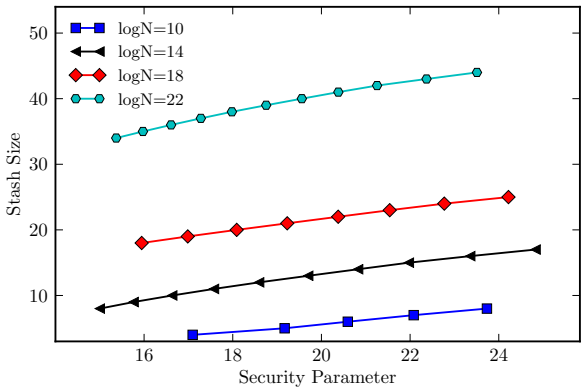


Figure 5: **Stash size of** SCORAM **grows linearly with security parameter.** $Z = 6$ and $\alpha = 4$.

5. OPTIMIZATIONS

Our optimizations largely fall into two categories: (1) determining the best parameters for SCORAM (Section 5.1); (2) improving the circuit design for frequently used logic components (Section 5.2).

In general, ORAM schemes have several decisive parameters including bucket size, stash size (if it uses a stash) and recursion factor. Depending on the specific eviction algorithms employed, there are additional parameters describing the eviction process. For example, in SCORAM, we use α to denote the number times to call flush. These parameters are subtly inter-related. For instance, a larger bucket size allows for smaller α and smaller stash at the same security parameter.

5.1 Parameter Optimization

We now systematically explore this parameter space to determine heuristically good choices for SCORAM.

Methodology. Some of our optimizations rely on simulations to count, for any particular ORAM setup, the number of ORAM failures (for estimating the security guarantee) and the total number of encryptions per memory access. We use simulations because all of the proofs that upper-bound the failure probabilities are too conservative in their approximations.

For each ORAM, we first run 16 million ORAM accesses to warm up the ORAM, such that it enters a steady state, we then start collecting numbers to determine the security parameters associated with this setup (including e.g., various bucket and stash sizes). Since the time average is equal to ensemble average for regenerative processes [13], we simulate each ORAM setup for a single long run of 1 billion accesses to estimate the security parameter (instead of multiple runs). This allows us to estimate ORAM parameters that achieve up to 2^{-80} security. A similar approach was suggested by Stefanov *et al.* [34].

Number of flushes and bucket size. We have run simulations of our new ORAM that indicate a good choice of α is 4.

After α is fixed, we consider two strategies to configure bucket size: 1) a uniform bucket size everywhere; and 2) varying bucket sizes across levels. For the former, we empirically determine an optimal bucket size of 6. However, we observe that, in SCORAM, buckets at the middle and lower part of the path tend to be more congested. This motivates us to redistribute the bucket size across levels. For example, for a binary tree of 21 levels, we find that increasing the bucket size by 1 for the first 10 buckets from root and decreasing the bucket size by 1 for the last 10 bucket at leaf is a better distribution than evenly distributing buckets. Assuming 80-bit security, varying the bucket size in this way allows us to reduce the stash size from 66 to 50, resulting in a circuit of size of 4,094,832 gates versus 4,562,988 for the version of SCORAM reported in Table 1, i.e., roughly a 10% reduction.

Stash size. We plot the stash size versus security parameters with different $\log N$ in Figure 5. Each point (x, y) on the curve should be read as "with a stash size of more than y, ORAM failures were observed 2^{-x} fraction of the time". the stash size grows linearly with security parameters, suggesting the failure probability decreases exponentially with the stash size. Note that unlike the Path-SC ORAM, the stash size is super-linear in $\log N$.

χ	ℓ	Total # gates $N = 2^{20}$	Total # gates $N = 2^{29}$
2	9	6,657,629	20,363,468
4	5	**4,562,988**	13,619,451
8	3	4,657,921	**13,382,510**
16	2	5,518,948	15,657,910
32	1	6,933,117	21,455,341
64	1	11,120,697	34,165,313

Table 2: **How to pick χ and the recursion level.** This table shows how we concretely optimize recursion parameters needed to implement the position maps in SCORAM. χ describes how many addresses are packed into each block, and ℓ describes the number of recursive steps before we use the linear-scan ORAM as the base case.

Recursion factor. Here we study the choices needed to recursively implement the position-map in all tree-based ORAM designs. We formulate the choice as an optimization problem as follows.

As before, let N denote the number of blocks (and thus the size of the position map), D denote the number of bit per blocks, and for our given scheme, let $f(N, D) = O(D \log^e N)\omega(1)$ denote the the circuit size for one ORAM access *excluding* the cost of the position map lookup. Let $LS(N, D)$ denote the circuit size of the linear scan ORAM.

In our top-level ORAM, we have $D_0 = 32$. When implementing the position map recursively with another ORAM, we must select the number of labels, χ, to pack into a block and the number of times, ℓ, to do recursion before finally using the linear scan ORAM as the base case. Note, the ORAM for the i^{th} recursive level has $N_i = \frac{N}{\chi^i}$ blocks of size $D_i = \chi \log N_{i-1}$. Thus, the total cost with recursion is

$$\sum_{i=0}^{\ell} f(N_i, D_i) + LS(N_{\ell+1}, D_{\ell+1})$$

For example, when $N = 2^{20}$, $\chi = 8$ and $\ell = 3$, we have total cost: $f(2^{20}, 32)+f(2^{17}, 160)+f(2^{14}, 136)+f(2^{11}, 112)+LS(2^8, 88)$. Since we can empirically determine f for any parameter setting, we can thus minimize the total cost. In our case, we limit χ to a power of 2 and use the same χ at each recursive level. See Table 2 for the results of this optimization. When $N = 2^{20}$, we use $\chi = 4$ and $\ell = 5$, and when $N = 2^{29}$, we use $\chi = 8$ and $\ell = 3$.

5.2 Circuit-Level Optimizations

We make several circuit-level optimizations to reduce the total number of gates and number of non-free gates. The latter (i.e., non-free gates) is a metric specific to certain backends such as garbled circuits [39] and GMW [8].

Backend-independent optimizations. A frequently-used logic is to determine if a block is dummy. We add a single bit field isDummy to each block indicating whether the block is `dummy`. This simple trick reduces the circuit size to determine if a block is dummy from $\log N$ AND gates to 1 AND gate. In addition, an important side benefit is that it enables efficient oblivious removal of a block from the bucket.

We only need to set the isDummy field instead of resetting all bits of a block.

Backend-dependent optimizations. Some secure computation protocols such as the garbled circuit and the GMW protocol support almost-free XOR gates. We make several circuit level optimizations exploiting this opportunity.

A common operation in some Eviction algorithms (e.g., CLP ORAM, SCORAM) is to compute the block among a given bucket of Z blocks $\{b_1, \ldots, b_Z\}$ that can be pushed deepest along the path P without violating the path invariant. Intuitively, we can calculate $\mathsf{LCA}(b_i.\mathsf{label}, P)$ for every i, which involves counting number of zeros of $(b_i.\mathsf{label} \oplus P)$. Since each label has $\log N$ bits, counting the number of leading zeros for every label requires $O(\log N \log\log N)$ AND gates and computing the maximum for all Z blocks requires $O(Z \log\log N)$ AND gates. Therefore, it uses a total of $O(Z \log N \log\log N)$ AND gates.

In our implementation, we use a circuit of size $O(Z \log N)$ to compute exactly this function: (1) compute $\mathsf{label}'_i = b_i.\mathsf{label} \oplus P$.(More zeros in label' means deeper). (2) in label', set all bits lower than the most significant 1 bit to 1. (3) find the block with smallest label', which corresponds to the deepest block. This works because a label' is smaller after processed by (2) if and only if it has more zeros.

6. PERFORMANCE EVALUATION

6.1 Methodology and Metrics

Our evaluation focuses on the following types of metrics:

1. *Cryptographic backend independent* metrics, such as gate count. This characterizes the performance of a ORAM when used in secure computation in general, relatively independent of the cryptographic backends.
2. *Cryptographic backend dependent* metrics, such as the number of encryptions, non-free gates, and bandwidth. These metrics characterize the performance of an ORAM scheme with a semi-honest Garbled Circuit backend in a manner that is independent of the specific hardware configuration or implementation artifacts.
3. *Implementation and machine dependent* metrics, such as runtime and breakdown of runtime. We will describe our specific hardware configuration and the specific Garbled Circuit implementation as the context of our results. We also discuss the interpretation of these results and project the performance had the experiments been run on a different hardware configuration or with a more optimized Garbled Circuit implementation.

6.2 Performance Comparison

We reevaluate the state-of-the-art ORAMs over secure computation, and compare their performance with our SCORAM. The metrics we considered include total gate count (Figure 6a), non-free gate count (Figure 6b), number of AES encryptions (Figure 6c), and input size of circuit (Figure 6d).

Table 3 summarizes the margin by which our SCORAM outperforms existing schemes. In particular, we show that across all metrics, we achieve 7.6× to 9.8× performance improvements comparing to the respective second best ones.

As mentioned earlier, even though Path-SC ORAM is asymptotically better than Binary Tree ORAM, for practical ranges of N, Binary Tree ORAM outperforms Path-SC ORAM due to lower constants in the asymptotic bound. This is shown in Figure 6.

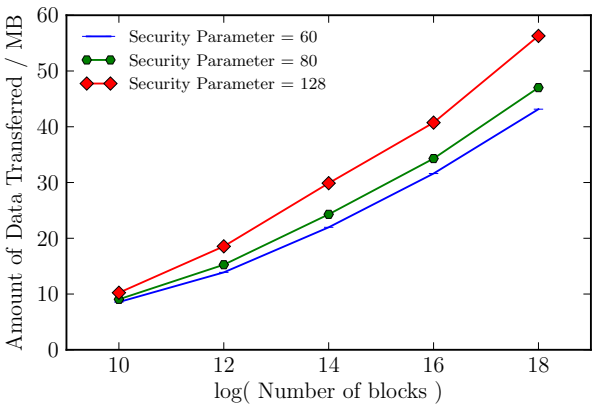


Figure 8: **The amount of data transferred per access for** SCORAM. Payload bitlength is 32 bit, 80-bit security parameter.

The most popular ORAM scheme used previously is Binary Tree ORAM. SCORAM is 7 times smaller and requires 27x fewer inputs. Note that each input bit involves an oblivious transfer, which, even with OT extension, incurs 2 encryptions. For the sake of reproducibility, we report all parameter settings that we used to run our experiments in Table 4 in the Appendix.

6.3 Performance Profiling for SCORAM

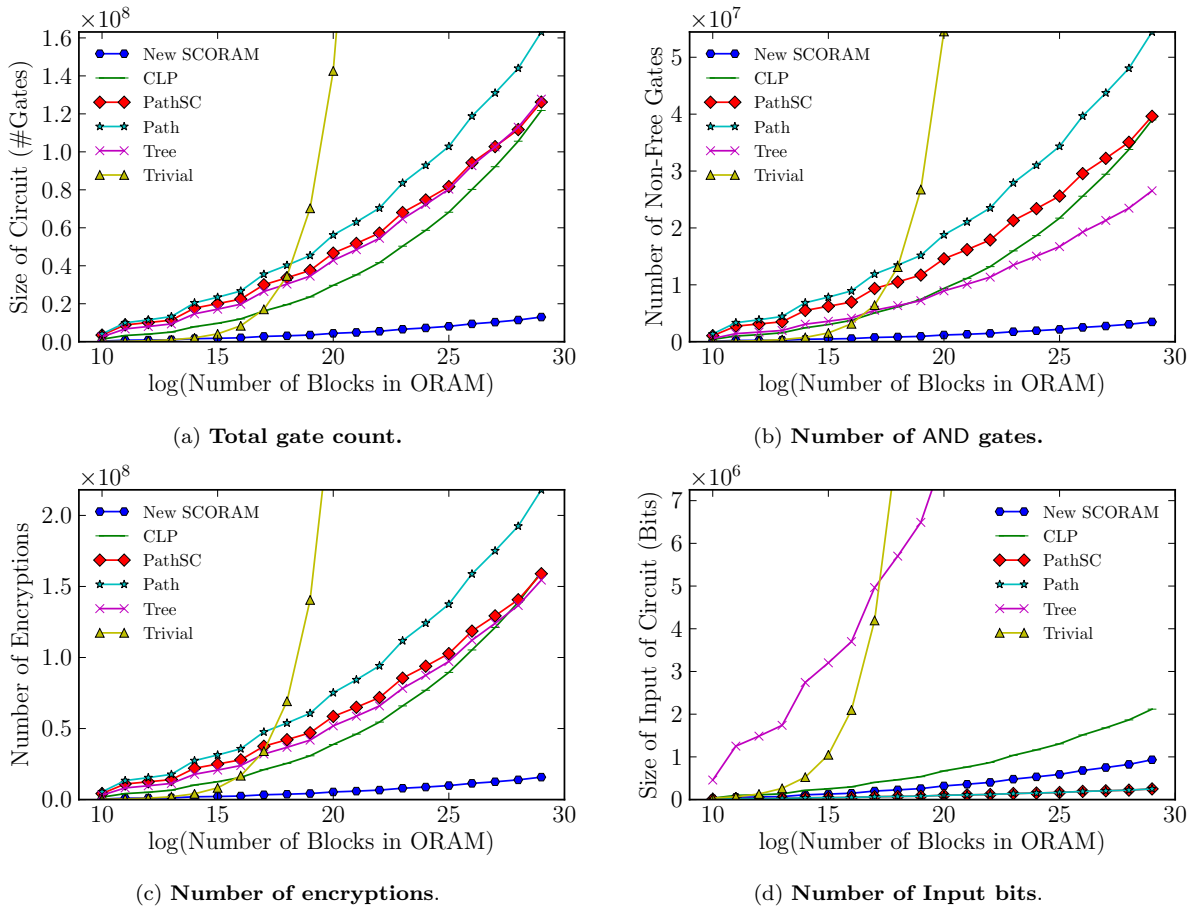
We further investigate the performance breakdown of SCORAM scheme. Specifically, the cost can be broken down into I/O overhead and computation overhead.

Our machine/implementation-dependent measurements are taken on a single server (Intel Xeon 2.13G Hz) running the circuit generator and evaluator as two independent processes (communicating through Socket). The memory usage ranges from 4–12 GB for both processes depending on the data size.

Interpreting the measurements. First, our implementation does not currently exploit AES-NI instructions to speed-up garbling. We expect a noticeable speedup for the computation time when hardware AES is implemented.

Second, the Garbled Circuit backend we used is a Java-based implementation. Therefore, our timing measurements are subject to the artifacts of the Java-based implementation. There are two main sources of artifacts, memory garbage collector and I/O synchronization. In Figure 7, we note that a substantial portion of the time is due to I/O. Since the experiments is run on the same machine where network bandwidth is not a bottleneck, we infer majority of the I/O time we recorded are due to I/O synchronization. Note that for the circuit generator, we observe the computation time spent on OT and garbling is roughly the same. On the circuit evaluator side, the portion of I/O cost is larger because the evaluator indeed has less computational work to do (thanks to the garbled row reduction technique) while being stuck more often waiting for the generator.

Network transmission. Figure 8 plots the bandwidth consumption for our SCORAM, under different security parameters. Note that the bandwidth overhead is proportional



(a) **Total gate count.**

(b) **Number of AND gates.**

(c) **Number of encryptions**.

(d) **Number of Input bits**.

Figure 6: **Comparison of various ORAMs.** Payload bitlength = 32 bits, security parameter = 80.

to the total number of non-free gates in our garbled circuit system. The non-free-gate counts (Figure 6b) correlates with growth in the bandwidth.

Comparison with existing implementations. We are aware of two other ORAM implementations over secure computation. Gordon *et al.* [12] report on an ORAM with $N = 2^{20}$ and $D = 512$ with secure parameter less than 20. The paper does not clearly report the bucket size they use, but we assume the bucket size is 40 based on our interpretation of their paper. Their ORAM requires 11.9M gates and takes approximately 50 seconds for one operation. In contrast, our best implementation for their parameters requires 3,743,213 total gates, and our implementation at significantly higher security parameters(80) runs in under 30 seconds. Further, Gordon *et al.* reported a break-even point with the trivial Linear Scan ORAM at 2^{16} blocks with 512 bits per block and security parameter less than 20. In comparison, we achieve a break-even point at 2^{14} blocks with 32 bits per block, with a much higher security parameter of 80. Note that typically, bigger blocks would reduce the break-even point because meta-data operations that are independent of block size can be amortized to more bits per block.

The second implementation is by Keller and Scholl [15]. Their implementation was based on the SPDZ protocol, running in the *preprocessing* model. Their flagship scheme was based on Path ORAM, with heuristic modifications. However, their result shows that their new Path ORAM variant is worse than Binary Tree ORAM, while our implementation is much better than Binary Tree ORAM. We expect that SCORAM would also outperform Binary Tree ORAM when implemented over SPDZ.

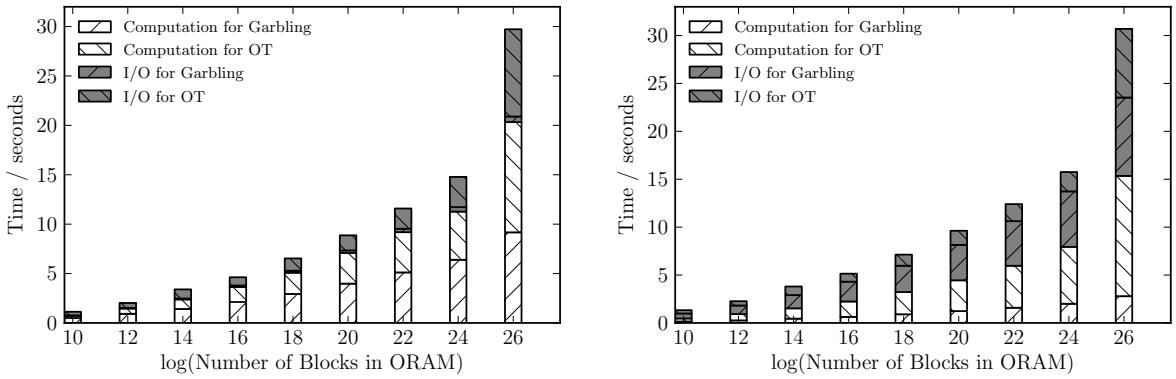
6.4 End-to-End Applications

In this section, we show the performance of SCORAM when used in practical applications. We assume Alice holds a sorted array and Bob holds queries. They would like to repeatedly search for some element over the sorted array without revealing any information except the result. The array is put into an ORAM in a one-time offline setup phase, and henceforth the parties will securely evaluate binary searches on multiple queries. Whenever the binary search program accesses a cell of the array, an ORAM access is called. In Figure 9, we show the clock time to do a binary search with different data size and payload size. The evaluation here reports the online cost of performing the searches, not counting the offline ORAM setup phase.

Gordon *et al.* [12] also implement a binary search application. However, they report performance only for a security parameter of 20. To achieve a security parameter of 80, their bucket size choice must be 4 times larger. Based on this, we estimate that when both schemes are parametrized with a security parameter of 80 and 512 bits payload, our imple-

Metric	Second best	Performance gain of SCORAM $N = 2^{20}$	Performance gain of SCORAM $N = 2^{29}$
Gate Count	CLP ORAM	$6.7X$	$9.4X$
Non-Free Gates	Binary Tree ORAM	$7.6X$	$7.6X$
Number of Encryptions	CLP ORAM	$7.2X$	–
	Binary Tree ORAM	–	$9.8X$

Table 3: **Performance comparison of SCORAM over the second best candidates.** (Data payload: 32 bits. Security parameter: = 80). Considering the number of encryptions, the second best is the CLP ORAM when $N = 2^{20}$, and the Binary Tree ORAM when $N = 2^{29}$.



(a) Breakdown for the Garbled Circuit *generator*.

(b) Breakdown for the Garbled Circuit *evaluator*.

Figure 7: **Cost breakdown of SCORAM.** (Data payload: 32-bit, Security parameter: 80-bit)

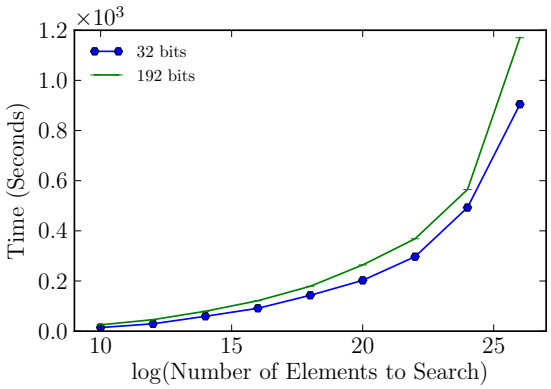


Figure 9: **Time to do a binary search over SCORAM in secure computation.** The result is shown with 80 bits security.

mentation is about 16x times faster than that by Gordon *et al.* [12]. Specifically, our SCORAM has about 8x times fewer AND gates; and our garbled circuit backend (including hardware configuration) is about 2x faster than Gordon *et al.*'s backend implementation and hardware.

7. CONCLUSION

As an enabling primitive of RAM-based secure computation, the construction of efficient ORAM for secure computation is of great importance. We are the first ones to observe that ORAM for secure computation requires a different metric than popular existing metrics for ORAM. We perform a thorough re-evaluation of state-of-the-art ORAM schemes based on a new circuit complexity metric. Further, we proposed a new, heuristic scheme called SCORAM, that empirically reduces the circuit size by 8X-10X compared with all existing ORAMs. SCORAM code will be released in the near future on our project webpage `http://www.oblivm.com`.

Acknowledgments. We would like to thank Marten van Dijk, Dov Gordon, Jonathan Katz, Kartik Nayak, and Saba Eskandarian for numerous helpful discussions. We gratefully acknowledge the anonymous reviewers for their insightful comments and suggestions. This work is supported by several funding agencies that are acknowledged separately on the front page.

8. REFERENCES

[1] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *ACM Conference on Computer and Communications Security*, 2008.

[2] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`, 2011.

[3] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. *arXiv preprint arXiv:1307.3699*, 2013.

[4] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[5] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[6] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[7] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[8] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM symposium on Theory of computing (STOC)*, 1987.

[9] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW*, 2011.

[11] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[12] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, pages 513–524, 2012.

[13] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, 2013.

[14] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*, 2011.

[15] M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. Cryptology ePrint Archive, Report 2014/137, 2014. `http://eprint.iacr.org/`.

[16] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *USENIX Security Symposium*, 2013.

[17] B. Kreuter, A. Shelat, and C. hao Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *USENIX Security Symposium*, 2012.

[18] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[19] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient ram-model secure computation. *IEEE S & P*, 2014.

[20] S. Lu and R. Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, pages 377–396, Berlin, Heidelberg, 2013. Springer-Verlag.

[21] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[22] P. MacKenzie, A. Oprea, and M. Reiter. Automatic Generation of Two-party Computations. In *ACM Conference on Computer and Communications Security*, 2003.

[23] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: A secure two-party computation system. In *USENIX Security*, 2004.

[24] J. C. Mitchell and J. Zimmerman. Data-Oblivious Data Structures. In *STACS 2014*, pages 554–565, 2014.

[25] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *CCS*, pages 801–812, 2013.

[26] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, 1990.

[27] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[28] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[29] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. *IEEE S & P*, 2014.

[30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[31] E. Stefanov and E. Shi. Fastprp: Fast pseudo-random permutations for small domains. Cryptology ePrint Archive, 2012. `http://eprint.iacr.org/`.

[32] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *CCS*, 2013.

[33] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.

[34] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *In CCS*, 2013.

[35] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *CCS*, 2014.

[36] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[37] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.

[38] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

[39] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.

APPENDIX

A. A LOGARITHMIC DEPTH CIRCUIT

We describe how to construct a low-depth circuit for Path-SC ORAM. The main steps of the algorithm are:

1. **Setting temporary offset fields.** As described before, we first sort the array A obliviously according to the field bucket. For each entry $A[i]$, if it is a dummy block, set its field $A[i].\mathsf{offset} := \perp$. Otherwise, for $1 \le i \le LZ$, set $A[i].\mathsf{offset} := Z \cdot A[i].\mathsf{bucket} + 1$; recall that our final goal is to rearrange blocks such that the entry $A[i]$ should go to bucket $\lfloor \frac{i-1}{Z} \rfloor$, and so the offset value is the smallest index at which the block can reside in A. For $LZ + 1 \le i \le LZ + \mathsf{stsize}$, the block $A[i]$ cannot be written back in the path P, and we set $A[i].\mathsf{offset} := \mathsf{st}$. Setting the offset fields takes circuit of size $O(S)$ and depth $O(1)$. For sorting the key values, we use the natural order for positive integers $\mathbb{Z}^+$, and use the convention $\mathbb{Z}^+ < \mathsf{st} < \perp$. Observe that for $\mathsf{stsize} + LZ + 1 \le i \le \mathsf{stsize} + 2LZ$, $A[i]$ must contain a dummy block.
2. **Determining final offset of real blocks.** The blocks that are going to be written back in the path P can come from only $A[1..LZ]$, which is sorted according to the field offset, that currently indicates the lowest index at which the block can reside in A. The purpose of this step is to update the offset field to the final position of each real block within A before being written back to the path P. This can be achieved by linear scan as described in Figure 4 in Section 3.1. However, a naive implementation will incur a circuit depth of $\Theta(LZ)$. We shall describe a more careful implementation using divide and conquer with circuit depth of $O(\log LZ)$.

A naive way to implement linear scan will incur a circuit depth of $\Theta(LZ) = \Theta(\log N)$. We shall describe a more careful implementation such that the circuit depth is dominated by that for oblivious sorting. Recall the input of the problem is an array $A[1 \dots m]$ that is sorted according to the offset field, which indicates the smallest integer that can be received by the entry. The problem is equivalent to increasing the offset field of each entry as little as possible such that the array A is still sorted according to offset and no two entries have the same offset value.

The high level idea is to use divide and conquer. The standard procedure is to first solve the problem recursively on $A[1..j]$ and $A[j+1..m]$, where $j = \lfloor \frac{m}{2} \rfloor$. Observe that the offset fields of entries in $A[1..j]$ do not have to be changed, and in order to achieve $O(1)$ circuit depth, we need to update the entries $A[j+1..m]$ in parallel.

Observe that at this point, the next available integer for $A[j+1]$ is $p := A[j].\mathsf{offset} + 1$. Hence, $r := \max\{p - A[j+1].\mathsf{offset}, 0\}$ is the amount that we need to increase $A[j+1].\mathsf{offset}$. The issue is whether we are able to deduce the increment readily for entries $A[i]$, for all $j + 1 \le i \le m$.

For such an i, observe that $r_i := (A[i].\mathsf{offset} - A[j+1].\mathsf{offset}) - (i - j - 1)$ is the number of integers in $[A[j+1].\mathsf{offset}..A[i].\mathsf{offset}]$ that have not been assigned to any entry. Hence, the amount we need to increase $A[i].\mathsf{offset}$ is $\max\{r - r_i, 0\}$. Hence, this step can be done in parallel for all $j + 1 \le i \le m$. The whole procedure is achieved by calling $\mathsf{UpdateOffset}(A[1..m])$, whose pseudocode is given in Algorithm 2. A standard analysis shows that this leads to a circuit of size $O(m \log m)$ and depth $O(\log m)$.

Algorithm 2 $\mathsf{UpdateOffset}(A[a..b])$

```
if a=b then return;
j := ⌊(a+b)/2⌋;
UpdateOffset(A[a..j]);
UpdateOffset(A[j+1..b]);
r := max{A[j].offset + 1 − A[j+1].offset, 0};
s := A[j+1].offset;
for i from j+1 to b in parallel do
    A[i].offset := A[i].offset+
      max{r − (A[i].offset − s) + (i − j − 1), 0};
return;
```

B. A DESCRIPTION OF ALL EXPERIMENTS

Figure 4 includes detailed parameters used in this paper in order to facilitate reproducible experiments.

ORAM design	N	Other Parameters Z	Stash, Evict, ℓ	Gates (M)	Inputs (M)
Binary Tree ORAM	2^{20}	120	N/A,2, 4	38.5	7.0
	2^{29}	120	N/A,2, 8	127.7	24.2
CLP ORAM	2^{20}	4	120,2, 4	29.4	0.7
	2^{29}	4	144,2, 8	121.8	2.1
Naive Path ORAM	2^{20}	4	89,1, 4	56.2	0.1
	2^{29}	4	89,1, 8	163.1	0.3
Path-SC ORAM	2^{20}	4	89,1, 4	37.2	0.1
	2^{29}	4	89,1, 8	111.7	0.3
SCORAM	2^{20}	6	88,4, 4	4.4	0.3
	2^{29}	6	141,4,8	13.0	0.9

Table 4: **A listing of all parameters used in our experiments.** The parameters are set to achieve statistical security of 2^{-80}. Gates and Inputs are obtained with payload bitlength = 32bits; Cutoff threshold is set at 2^{10}