A Tutorial on Oblivious Random Access Memory

Elaine Shi and Srinivas Devadas

***Abstract*—Oblivious RAM (ORAM) is a key technique for protecting memory access patterns in systems with untrusted memory, including secure processors, trusted execution environments, and multi-party computation. This article presents a tutorial overview of ORAM concepts and designs from an algorithmic and cryptographic perspective. We review the security goals and threat models underlying ORAM, describe the most popular ORAM constructions and their core mechanisms, and discuss practical considerations for implementing ORAM. The tutorial aims to provide readers with a clear foundation for understanding ORAM-based memory protection and its applications.**

I. What is Oblivious RAM?

Imagine that a user or a company wants to store some secret data on some untrusted cloud provider. While the data contents can be protected through encryption, the user is worried that accesses to the (encrypted) data can leak sensitive information. Below are some simple examples of access pattern leakage:

- Imagine the user outsources encrypted genomic data to the cloud, and a doctor retrieves certain snippets as part of a genetic screening test. If the server observes which snippets are fetched, it can potentially infer what disease the user is being screened for.
- Imagine that a user outsources an encrypted database consisting of sorted records to the server. The user then performs a binary search on the records. It is not hard to see that the entries accessed during the search could leak information about the query.
- More generally, we can view access pattern leakage through a programming language perspective: the following program has an `if`-branch dependent on secret inputs (e.g., think of the secret input as the last bit of a secret key) Thus, by observing whether memory location `x` or `y` is accessed, one can infer which branch is taken.

```
if (s) {
     mem[x]
} else {
     mem[y]
}
```

A second scenario is secure processors which are often used for confidential computation on sensitive data. A secure processor provides an enclave whose security is enforced through the hardware. Data is normally encrypted at rest or in transit, and the secure enclave internally has a secret key that can decrypt the data and perform computation inside. However, today's off-the-shelf secure processors such as Intel's SGX do not protect access patterns to (encrypted) data. Thus, an adversary monitoring the memory access patterns can discover secret or sensitive information without looking inside of the enclave.

With these motivating examples in mind, we ask the following fundamental question:

How can we provably hide access patterns while preserving efficiency?

A. Oblivious RAM: Problem Definitions

Oblivious RAM (ORAM), first proposed in the seminal work by Goldreich and Ostrovsky [1], [2], is a powerful cryptographic protocol that *provably* hides access patterns to sensitive data.

We would like to ensure a very strong notion of security. In particular, no information should be leaked about: 1) which data block is being accessed; 2) the age of the data block (i.e., when it was last accessed); 3) whether a single block is being requested repeatedly (i.e., frequency); 4) whether data blocks are often being accessed together (i.e., co-occurrence); or 5) whether each access is a read or a write.

An ORAM is an algorithm that converts *logical* read and write requests from a client to *physical* requests to memory or cloud server storage (see Figure 1). To formally define ORAM and its security, we henceforth refer to each atomic memory word as a memory *block*, and we use the notation N to denote the total number of blocks of the logical memory.

An ORAM algorithm (also referred to as the *client*) receives as inputs a sequence of *logical* requests where each logical request is of the form

$$(\texttt{read}, \texttt{addr}) \text{ or } (\texttt{write}, \texttt{addr}, \texttt{data}).$$

Upon receiving each request, an ORAM algorithm may interact with a "memory" (also referred to as the *server*) to make a sequence of *physical* accesses where each physical access either reads or writes a physical location, at the end of which the ORAM algorithm returns an answer to the logical input request. The adversary (i.e., server) can observe this physical access sequence, but cannot directly observe the logical input requests or the private random coins used by the ORAM algorithm.

Given a logical request sequence X, the random variable $\mathsf{Addresses}(X)$ denotes the physical accesses (including the addresses as well as whether each access is a read or write) resulting from the ORAM algorithm for the input sequence X. We say that an ORAM scheme is secure iff for any N and for any two logical request sequences X_0 and X_1 of the same length,

$$\mathsf{Addresses}(X_0) \approx \mathsf{Addresses}(X_1).$$

Parts of this tutorial (with modifications) have been published as a part of the "Introduction to Programmable Cryptography" textbook, an effort by 0xPARC.

E. Shi is with the Department of Computer Science, Carnegie Mellon University, Pittsburgh (e-mail: elaineshi@cmu.edu).

S. Devadas is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge (e-mail: devadas@mit.edu).

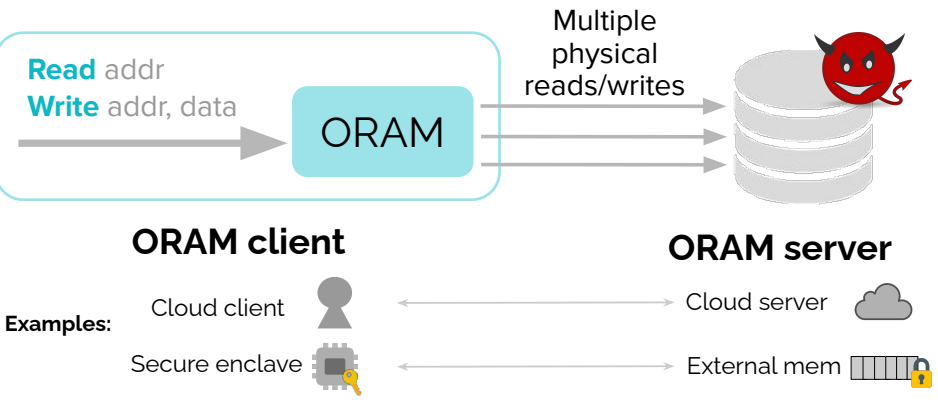


Fig. 1: ORAM can hide access-pattern leakage in both cloud outsourcing and secure-processor settings. In the cloud-outsourcing setting, ORAM runs on the trusted client and translates each logical request into multiple physical accesses to an untrusted server. In the secure-processor setting, ORAM runs within a trusted enclave, while external memory is untrusted.

Specifically, $\approx$ can denote *statistical* or *computational* indistinguishability (see Section 3.1 of Pass and Shelat's textbook [3]). The resulting ORAMs are called *statistically secure* or *computationally secure*, respectively. Intuitively, our security definition requires that for any two *logical* request sequences, the ORAM's resulting *physical* access sequences are indistinguishable.

An ORAM algorithm's cost can be measured by the (average) number of physical accesses needed to serve each logical request. For most of the tutorial, we adopt a simple word-RAM machine model where each memory block is a single word — this is also the default model of computation used in standard undergraduate-level algorithms textbooks [4], [5]. Later in Section VI-B, we will survey performance metrics depending on the application scenario and computational model.

Remark 1. We make some additional remarks about the obliviousness notion.

1) In our definition of Addresses, i.e., what the adversary can observe, we did not include the contents of the memory blocks, only the physical addresses and whether each physical access is a read or write. In practice, we may use encryption or secret sharing to hide the contents of the blocks.

   One may ask why statistical security is interesting if we use encryption anyway to hide the data contents. In fact, besides the theoretical interest, there is an important practical reason: if the ORAM scheme is employed in a multi-party computation (MPC) protocol, we would not have to securely evaluate cryptographic primitives (e.g., pseudo-random functions) with MPC, see Section II-F.
2) Whenever we use statistical or computational indistinguishability, we implicitly assume some security parameter λ. Statistical indistinguishability means the statistical distance of the two distributions must be upper bounded by some negligible function in the security parameter λ. Computational indistinguishability is defined as in the standard cryptography literature. Since we are concerned about the asymptotical performance for large N, without loss of generality, we may assume that $N \geq \lambda$.
3) Some literature defines a stronger security notion called adaptive security [6]. Adaptive security allows the adversary to choose the next request after having observed the outcomes and access patterns of previous requests. In this tutorial, for simplicity, we use the static security notion. The terminology adaptive/static security should not be confused with online/offline ORAM. By default, an ORAM is an online data structure: it processes requests as they arrive without knowing the entire request sequence in advance. Offline ORAM, by contrast, has been studied as an intermediate theoretical abstraction and is more akin to oblivious sorting [1], [7].

B. Naïve Solutions

Naïve solution 1. One trivial solution is for the client to read all blocks from the server upon every logical request. Obviously, this scheme leaks nothing but would be prohibitively expensive.

Naïve solution 2. Another trivial solution is for the client to store all blocks, and thus the client need not access the server to answer any memory request. But this defeats the numerous advantages of cloud outsourcing in the first place. *Henceforth, we require that client store only a small amount of blocks* (e.g., constant or polylogarithmic in N).

Naïve solution 3. Another naïve idea is to randomly permute all memory blocks through a secret permutation known only to the client. Whenever the client wishes to access a block, it will appear to the server to reside at a random location.

Indeed, this scheme gives a secure *one-time* ORAM scheme, i.e., it provides security if every block is accessed only once. However, if the client needs to access each block multiple times, then the access patterns will leak statistical information such as frequency (i.e., how often the same block is accessed)

and co-occurrence (i.e., how likely two blocks are accessed together). As mentioned earlier, one can leverage such statistical information to infer sensitive secrets [8].

Important observation. Naïve solution 3 gives us the following useful insight: informally, if we want a "non-trivial" ORAM scheme, it appears that we may have to relocate a block after it is accessed — otherwise, if the next access to the same block goes back to the same location, we can thus leak statistical information. It helps to keep this observation in mind when we describe some ORAM schemes later.

C. Brief History of ORAM

ORAM was first proposed by Goldreich and Ostrovsky [1], [2], who showed a computationally-secure construction with $O(\log^3 N)$ overhead assuming the existence of pseudorandom functions (PRFs). Goldreich and Ostrovsky's work established a general paradigm for constructing ORAMs, called the *hierarchical framework* in the subsequent literature. The hierarchical framework can be viewed as an application of Bentley and Saxe's classical method [9] of making static data structures dynamic.

The first statistically secure ORAM was proposed by Shi et al. [10], and the same work also established a new paradigm for constructing ORAMs, called the binary-tree framework. Besides achieving statistical security, the *binary-tree framework* also yields simple ORAM constructions with fast concrete performance. While the original binary-tree ORAM of Shi et al. [10] incurred $O(\log^3 N)$ overhead, the subsequent literature, notably Path ORAM [11] and Circuit ORAM [12] showed how to shave a logarithmic factor, improving the overhead to $O(\log^2 N)$.

Today, ORAM is deployed at a large scale in real-life applications. Notably, Signal uses ORAM as a backbone for its private contact discovery service [13]. Besides Signal, Meta is also planning to roll out new services using ORAM [14]. Section V provides more details on the real-world usage scenarios.

On the lower bound side, Goldreich and Ostrovsky [1] showed that any statistically secure ORAM must have at least $\Omega(\log N)$ overhead. Later, Larsen and Nielsen extended their lower bound to work even for computationally secure ORAMs [15]. For the computational security setting, we know how to construct an optimal ORAM matching the lower bound due to the work of Asharov et al. [16], which in turn builds upon ideas from Patel et al. [17].

In sections Section II and Section III, we shall describe the two frameworks, binary-tree and hierarchical, respectively, starting with the binary-tree framework, which is somewhat simpler to explain.

II. Binary-Tree-Based ORAMs

A. Data Structure

We will learn about tree-based ORAMs, first proposed by Shi et al. [10]. Tree-based ORAMs provide a framework to construct very simple ORAM schemes: in our tutorial, we will first describe the original binary-tree ORAM by Shi et al. [10]. Then, we will mention improvements called Path ORAM [11], [18], and Circuit ORAM [12] that made binary-tree ORAMs practical.

Server data structure. The server stores a binary tree (see Figure 2), where each node is called a *bucket*, and each bucket is a finite array that can hold up to Z number of blocks — for now, think of Z as being relatively small (e.g., polylogarithmic in N or constant); we will describe how to parametrize Z later. Some of the blocks stored by the server are *real*, i.e., they carry actual data; and others are *dummy*, i.e., they do not encode useful information and are only present for achieving security. All Z blocks in a bucket are read or written in a read or write, respectively.

Main path invariant. The most important invariant is that at any point of time, each block is mapped to a random path in the tree (also referred to as the block's designated path), where a path begins from the root and ends at some leaf node — and thus a path can be specified by the corresponding leaf node's identifier. When a block is mapped to a path, it means that the block can legitimately reside anywhere along the path (see Figure 2).

Imaginary position map. For the time being, we will rely on the following assumption, which we will get rid of later. We assume that the client can store a somewhat large position map that records the designated path of every block (see Figure 2). In general, such a position map would require roughly $\Theta(N \log N)$ bits to store — but later we can recursively outsource the storage of the position map to the server by placing them in progressively smaller ORAMs.

B. Operations

We now describe how to access blocks in our ORAM scheme.

Block details. A block is an (address, data) tuple that is encrypted, where the address is the logical address referred to by the client. Therefore, the server cannot see the data or the address. The server maintains the tree, and can follow paths from the leaves to the root, or vice versa.

Fetching a block. Given how our data structures are set up, accessing a block is very easy: the client simply looks up its local position map, finds out on which path the block is residing, and then reads each and every block on the path. As long as the main invariant is respected, the client is guaranteed to find the desired block.

Remapping a block. Recall that earlier, we have gained the informal insight that whenever a block is accessed, it should relocate. Here, whenever we access a block, we must remap it to a randomly chosen new path — otherwise, we would end up going back to the same path if the block is requested again, thus leaking statistical information.

To remap the block, we choose a fresh new path, and update the client's position map to associate the new path with the block. We now would like to write this block back to the tree,

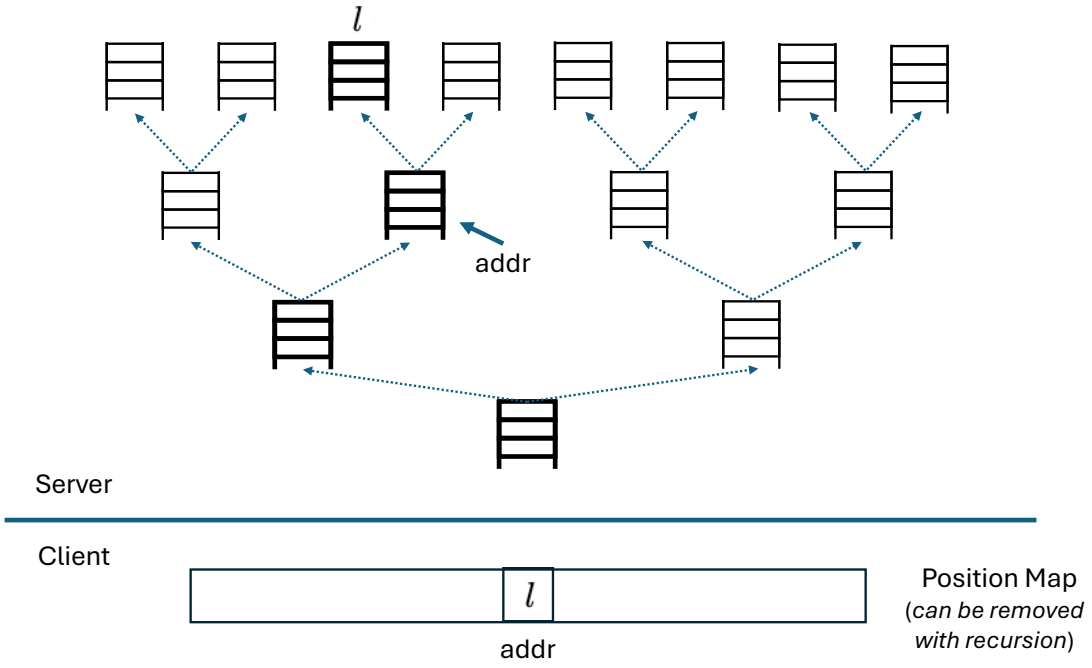


Fig. 2: ORAM's binary-tree data structure. Every logical address is assigned to a random path, and must reside on this path. Accessing the corresponding path will locate the desired element. One can imagine that the client has a position map that stores the paths associated with all addresses. This position map will be later removed through recursion.

to somewhere on the new path (and if the request is a `write` request, the block's contents are updated before being written back to the server). But doing this is tricky! It turns out that we cannot write the block back directly to the leaf bucket of the new path. Doing so can leak information which new path the block got assigned. For example, if the next request asks for the same block, it would then go to this new path; otherwise most likely the next request will go to a different path. By a similar reasoning, we cannot write this block back to any internal nodes of the new path either, since writing to any internal node on the new path also leaks partial information about the new path.

It turns out that the only safe location to write the block back is to the root bucket! The root bucket resides on every path, and thus writing the block back to the root does not violate the main path invariant; and further, it does not leak any information about the new path.

Our idea thus is to write this block back to the root bucket. However, there is an obvious problem. The root bucket has a capacity of Z, and if we keep writing blocks back to the root, soon enough the root bucket will overflow! Therefore, we now introduce a new procedure called *eviction* to cope with this problem.

Eviction. Eviction is a maintenance operation performed upon every data access to ensure that none of the buckets in the ORAM tree will ever overflow except with negligible in N failure probability. Note that if an overflow does happen, the block that leads to the overflow can get lost since there is no space to hold it on the server, and this can affect the correctness of our ORAM scheme. However, we will guarantee that such correctness failure happens only with negligible probability.

The high-level idea is very simple: whenever we can, we will try to move blocks in the tree closer to the leaves, to allow space to free up in smaller levels of the tree (i.e., levels closer to the root). There are a few important considerations when performing such eviction:

- Data movement during eviction must respect the main path invariant, i.e., each block can only be moved into buckets in which it can legitimately reside.
- Data movement during eviction must retain *obliviousness*, i.e., the physical locations accessed during eviction should be independent of the input requests to the ORAM.
- As we perform eviction, we pay a cost for this maintenance operation and the cost is charged to each data access. Obviously, if we are willing to pay more such cost, we can pack blocks closer to the leaves, thus leaving more room in smaller levels. In this way, overflows are less likely to happen. On the other hand, we also do not want the eviction cost to be too expensive. Therefore, another tricky issue is how we can design an eviction algorithm that achieves the best of both worlds: with a small number of eviction operations, we can avoid overflow almost surely (i.e., no overflow except with negligible in N probability).

We describe a simple candidate eviction scheme (see also Figure 3), and we will give an informal analysis of the scheme later:

[An eviction algorithm] Upon every data access, we choose 2 random buckets in every level of the tree for eviction (for

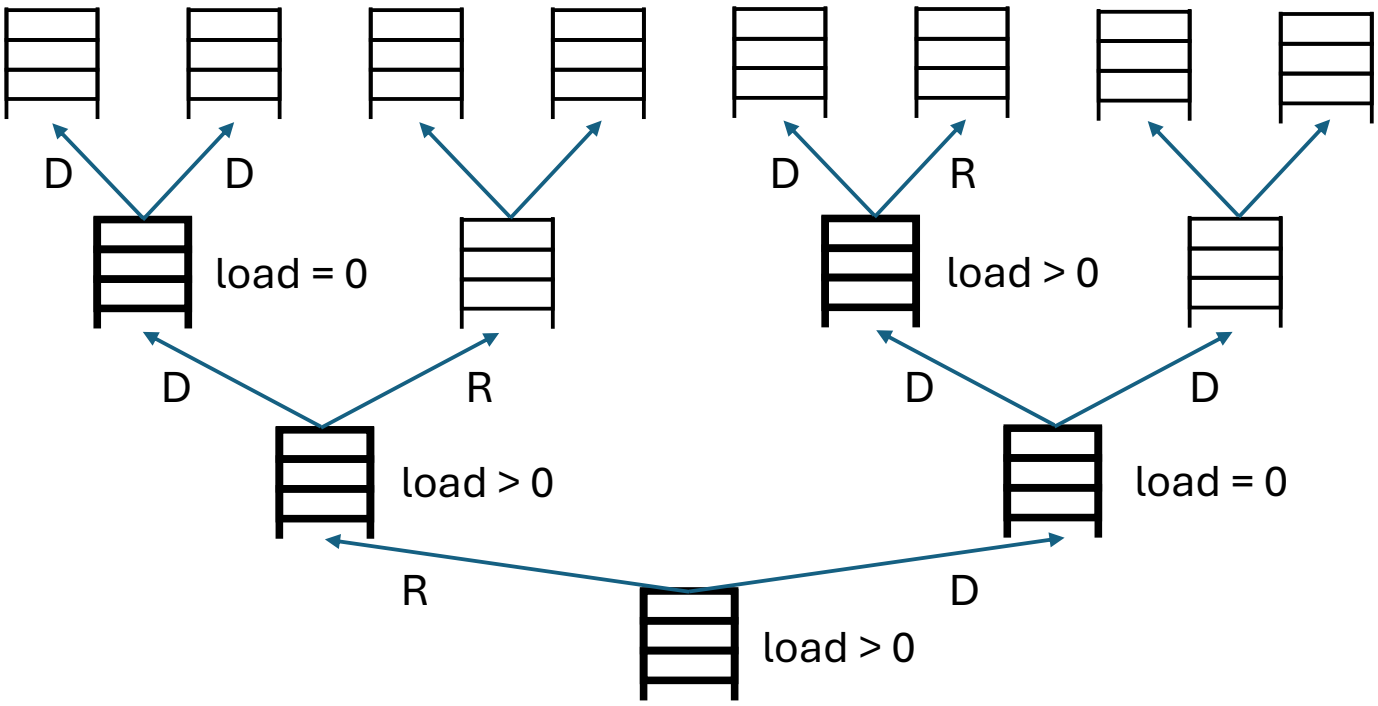


Fig. 3: The `Evict` algorithm. Upon every data access operation, 2 random buckets (shown in bold) are chosen at every level of the tree for eviction during which one data block will be evicted to one of its children. To ensure security, a dummy eviction is performed for the child that does not receive a block; further, if the bucket chosen for eviction is empty (i.e., only stores dummy blocks), dummy evictions are performed on both children buckets. In this figure, R denotes a real eviction and D denotes a dummy eviction.

the root level, pick one bucket). If a bucket is chosen for eviction, we will pop an arbitrary block (if one exists) from the bucket, and write the block to one of its children.

Note that depending on the chosen block's designated path, there is only one child where the block can legitimately go. We must take precautions to hide where this block is going: thus for the remaining child that does not receive a block, we can perform a "dummy" eviction. Additionally, if the bucket chosen for eviction is empty (i.e., does not contain any real blocks, only dummy blocks), then we make a dummy eviction for both children — this way we avoid leaking the information that the chosen bucket is empty.

More specifically, to write an intended block to a child bucket, we sequentially scan through the child bucket. If the slot is occupied with a real block, we simply write the block back. If the slot is empty, we write the intended block into that slot. A dummy eviction therefore is basically reading every block sequentially and writing the original contents back.

So far, we have not argued why any bucket that receives a block always has space for this block, i.e., there is no overflow except with negligible probability — we will give an informal analysis later to show that this is indeed the case (had it not been the case, the ORAM algorithm's correctness will be affected).

Algorithm pseudo-code. We present the algorithm's pseudo-code in Algorithms 1 and 2. The notation $\xleftarrow{\$}$ means sampling uniformly at random and $(\perp, \perp, \perp)$ denotes a dummy block where $\perp$ is a special string of appropriate length.

Algorithm 1: `Access`$(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ where $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$

1 **Assume:** each block is of the form $(\mathsf{addr}, \mathsf{data}, l)$, where l denotes the block's current designated path.
2 $l^* \xleftarrow{\$} [1..N]$; $l \leftarrow \texttt{position}[\mathsf{addr}]$; $\texttt{position}[\mathsf{addr}] \leftarrow l^*$
3 **foreach** *bucket from leaf l to root* **do**
4 Scan bucket: if $(\mathsf{addr}, \mathsf{data}_0, _) \in$ bucket then let $\mathsf{data}^* \leftarrow \mathsf{data}_0$ and remove this block from bucket (replace with a dummy block)
5 If $\mathsf{op} = \texttt{read}$: add $(\mathsf{addr}, \mathsf{data}^*, l^*)$ to the root bucket; else, add $(\mathsf{addr}, \mathsf{data}, l^*)$ to the root bucket
6 Call `Evict`()
7 **return** data^*

Remark 2. Note that in a full-fledged implementation, all blocks are typically encrypted to hide the contents of the block. Whenever reading and writing back a block, the block must be re-encrypted prior to being written back. If the encryption scheme is secure, the server should not be able to tell whether the block's contents have changed upon seeing the new ciphertext.

Algorithm 2: The procedure `Evict`

```
foreach level d from root to the level of leaves −1 do
    randomly choose two distinct buckets bucket_0 and bucket_1 in level d (for the root level, pick one bucket)
    for bucket ∈ {bucket_0, bucket_1} do
        block ← pop a real block from bucket if one exists; else, block ← (⊥, ⊥, ⊥)
        for each of the two children of bucket in a fixed order do
            scan the child bucket, reading and writing every block; if block is real and wants to go to this child then write block to an empty (i.e., dummy) slot in the child bucket
```

C. Analysis

We will now discuss why the aforementioned binary-tree ORAM construction 1) preserves obliviousness; and 2) is correct except with negligible in N probability.

Obliviousness. Obliviousness is in fact easy to see. First, whenever a block is accessed, it is assigned to a new path and the choice of the new path is kept secret from the server. Thus, whenever the block is accessed again, the server simply observes a random path being accessed. Second, observe that the entire eviction process does not depend on the input requests at all.

Correctness. Correctness is somewhat more tricky to argue. As mentioned earlier, to argue correctness, we must argue why no overflow will ever occur except with negligible probability — as long as the bucket size Z is set appropriately.

Claim II.1 (Bucket size and overflow probability)**.** *If the bucket size Z is $\omega(\log N)$, then over any polynomially many accesses, no bucket overflows except with negligible in N probability*[1].

Proof. Note that for the leaf nodes, we can apply a standard balls-and-bins analysis, that is, if we throw N balls into N bins at random, then by the Chernoff bound, we have that for any $k \in \mathbb{N}$,

$$\Pr[\text{max bin load} > k] \leq \exp(-\Omega(k)).$$

Henceforth we focus on analyzing non-leaf buckets. We shall first give a "buggy" proof, which is almost correct but to formalize it requires some extra work as explained later.

- First, observe that the root bucket (i.e., level 0 of the ORAM tree) receives exactly 1 incoming block with every access, but we get to evict the root bucket twice upon every access, and thus whatever enters the root gets evicted immediately. The root bucket is a special case and henceforth we no longer need to consider the root bucket in the analysis below.
- Now consider a bucket at level 1 of the ORAM tree. On average, one out of every two accesses, a block will enqueue in the bucket. With probability 1, the bucket will be chosen for eviction. If the bucket is chosen for eviction, a block gets to dequeue from this bucket.
- Similarly, now consider a bucket at level 2 of the ORAM tree. On average, one out of every four accesses, a block will enqueue in the bucket. With probability $\frac{1}{2}$, the bucket will be chosen for eviction.
- In general, we can conclude that for any non-leaf level $i > 1$ of the ORAM tree, with each access, one out of every 2^i accesses, a block will enqueue, and with probability $\frac{1}{2^{i-1}}$, the bucket is chosen for eviction.

Now we see a useful pattern: for every non-leaf and non-root level of the tree, with every ORAM access, the dequeue probability is twice as large as the enqueue probability. This reminds us of the standard discrete-time M/M/1 queue which you might have learned about in a basic probability class. In general, in a discrete-time M/M/1 queue,

- Every time step, with probability p, an item enqueues;
- Every time step, with probability $2p$, an item dequeues if the queue is non-empty.

Standard Markov chain analysis tells us that at any given time,

$$\Pr[\text{M/M/1 queue length} > R] \leq \exp(\Omega(-R)). \quad (1)$$

See Chapter 13 of Harchol-Balter's textbook for a proof of the above statement [19]. Thus, if each bucket indeed behaves like an M/M/1 queue, then Claim II.1 can be proven by simply applying Equation (1) to each bucket and then taking a union bound over all buckets and all time steps. In Equation (1), if we simply plug in any $R = \omega(\log N)$, the failure probability would be negligibly small in N. In other words, setting the bucket size $Z = \omega(\log N)$ gives a negligibly small overflow probability even after taking a union bound over all buckets and over $\mathsf{poly}(N)$ many requests.

Unfortunately, there is a bug in the above proof, as we previewed earlier. The reason is that the buckets in the ORAM tree are not independent, and our informal argument above ignored possible dependence between buckets. Well, fortunately, it turns out that this is not a big issue, and if we simply apply the discrete version of Burkes' theorem for tandem queues [20], we can in fact turn the above informal analysis into a formal proof! Imprecisely speaking, Burkes' theorem says that in such a tandem queuing system as the above, even though the queue lengths are not independent, it turns out that the *marginal* stationary distribution of each queue's length is the same as having independent M/M/1 queues. □

D. Binary-Tree ORAM: Recursion

So far, we have pretended that the client can store a large position map. We now describe how to get rid of this position map. The idea is simple: instead of storing the position map on the client side, we simply store it in a smaller ORAM

[1] Without loss of generality, we may assume that N is at least the security parameter, so negligible in N also means negligible in the security parameter.

denoted $\mathsf{posORAM}_1$ on the server. The position map of $\mathsf{posORAM}_1$ will then be stored in an even smaller ORAM denoted $\mathsf{posORAM}_2$ on the server, and so on. As long as the block size is at least $\Omega(\log N)$ bits, every time we recurse, the ORAM's size reduces by a constant factor; and thus $O(\log N)$ levels of recursion would suffice.

We can thus conclude with the following theorem.

Theorem II.2 (Binary-tree ORAM [10])**.** *For any super-constant function $\alpha(\cdot)$, there is an ORAM scheme that achieves $O(\alpha \log^3 N)$ cost for each access, i.e., each logical request will translate to $O(\alpha \log^3 N)$ physical accesses; and moreover, the client is required to store only $O(1)$ number of blocks.*

Note that in the total cost $O(\alpha \log^3 N)$, an $\alpha \log N$ factor comes from the bucket size; another $\log N$ factor comes from the total height of the tree; and the remaining $\log N$ factor comes from the recursion.

E. Path ORAM and Circuit ORAM

The design of the above binary-tree ORAM is inefficient: whenever we visit a triplet of buckets for eviction, we only evict one block. For this reason, the bucket size needs to be $\omega(\log N)$ to get negligible in N failure probability. The reason is that the original binary-tree ORAM was designed to fit the (relatively simple) proof the authors had in mind. It turns out that if we instead use a more aggressive eviction algorithm, and with a more sophisticated proof, the bucket size can be made constant. At this point, we are ready to introduce an improved version called Path ORAM.

Path ORAM. The Path ORAM algorithm, proposed by Stefanov et al. [11], [18], works as follows.

- *Data structure:* Unlike the above binary-tree ORAM, in Path ORAM, every bucket has constant size (e.g., 4 or 5) except the root bucket which is super-logarithmic in size.
- *Accessing a block.* To request a block, we first perform "read and remove" on a path just like the binary-tree ORAM. However, the eviction process is new. In Path ORAM, we perform eviction on the same path we have just read. In particular, we will rearrange the blocks on the path in the most aggressive manner possible:

  > **Path ORAM's eviction on a path:**
  > Move the blocks as close to the leaf level as possible, but *without violating the path invariant*.

- *Recursion.* We employ the same recursion technique to recursively store the position map.

With Path ORAM, ignoring the recursion, every access operation touches a single path, and hence the name Path ORAM. Assuming that the client has enough local space to store a single path, then Path ORAM enjoys $O(\alpha \cdot \log^2 N)$ cost per access (in terms of memory reads and writes) for an arbitarily small superconstant function α.

Circuit ORAM. Path ORAM requires that the client be able to store a path to get $O(\alpha \cdot \log^2 N)$ cost per access. If the client has only constant local space (e.g., applications in multi-party computation), Path ORAM's overhead becomes $O(\alpha \cdot \log^2 N \cdot \mathsf{poly} \log \log N)$ because the eviction on a path must be implemented using constant number of oblivious sorts — see also Section III-A.

A subsequent work called Circuit ORAM [12] improves Path ORAM to further reduce the reliance on client space. In Circuit ORAM, the eviction on a path is simplified such that eviction can be performed by making constant number of scans over the path's contents. Oblivious sorting on the path is no longer needed. As a results, Circuit ORAM achieves $O(\alpha \cdot \log^2 N)$ cost per request — in terms of both memory reads/writes and running time — even when the client has constant local space.

For practical implementations, either Path ORAM or Circuit ORAM are the schemes of choice (for moderately large data sets). Specifically, for cloud computing scenarios, Path ORAM or its variants such as Ring ORAM [21] are the schemes of choice. For multi-party computation [22], Circuit ORAM is the scheme of choice.

F. Advantages of Binary-Tree-Based ORAMs

Binary-tree-based ORAMs do not employ any cryptography. This leads to several advantages:

- They achieve a statistical notion of security.
- They enjoy fast concrete performance. In particular, binary-tree-based ORAMs enjoy performance comparable to common data structures, and there is no cryptographic overhead.
- They are desirable in multi-party computation (MPC) scenarios, since there is no need to evaluate any cryptographic primitives such as pseudorandom-functions (PRFs) wrapped inside MPC. For these reasons, hierarchical ORAMs (see Section III) which employ PRFs are not desirable in MPC settings.

We show some concrete performance results in Table I of Section VI-D.

Besides Path ORAM and Circuit ORAM, several other binary-tree-based ORAM variants have been proposed [23]–[25]. For example, the construction of Chung et al. [23], proposed around the same time as Path ORAM, was both asymptotically and concretely subsumed by Path ORAM. Gentry et al. [25] introduced a reverse-lexicographical eviction order, a practical optimization adopted by many subsequent works. Building on this eviction order, rORAM [24] explores how to achieve a locality-friendly memory layout.

III. Hierarchical ORAM

Chronologically, Hierarchical ORAM was the original ORAM construction, and was first proposed by Goldreich and Ostrovsky [1], [2]. Known ORAM constructions following the hierarchical paradigm rely on pseudorandom functions, and hence they achieve computational notions of security. The version we will describe in this tutorial is *a simplified and optimized variant* of Goldreich and Ostrovsky's original construction [1], [2] introduced in Chan et al. [6].

Since the hierarchical ORAM relies on some building blocks including oblivious sort and oblivious hash tables, we

shall first introduce these building blocks. In the rest of this section, we use the following notation:

- N denotes the space of the RAM that we want to make oblivious,
- n denotes the instance size for the underlying building blocks such as oblivious sorting and hash tables — in particular, in the final ORAM construction, these building blocks will be applied to instances of sizes smaller than N.

A. Building Block: Oblivious Sorting

Oblivious sorting receives an input array containing (key, value) pairs, and sorts the elements by their keys. The algorithm is said to be oblivious, iff for any two input arrays $\mathcal{I}$ and $\mathcal{I}'$ of the same length, the access patterns of the algorithm are computationally or statistically indistinguishable. For example, the commonly used QuickSort algorithm is not oblivious since it reads and writes arrays in a way that depends on the comparison result with some pivot.

Oblivious sort can be accomplished through sorting networks. A sorting network is a circuit with compare-and-swap gates. Each gate receives two elements, compares their keys, and either swaps them or doesn't swap them based on the comparison result. Well-known sorting networks include 1) bitonic sort [26], which sorts n elements with $O(n \log^2 n)$ compare-and-swap gates; and 2) the AKS sorting network [27], which sorts n elements with $O(n \log n)$ compare-and-swap gates. A sorting network with $T(n)$ compare-and-swap gates can be emulated on a RAM running in $T(n)$ time (assuming that the key and value can fit in constant number of words). It has also been shown that any oblivious sorting algorithm must incur $\Omega(n \log n)$ time on a RAM [28], [29] (under reasonable assumptions). In this sense, the AKS algorithm is asymptotically optimal.

Unfortunately, the AKS algorithm is complex and concretely inefficient due to the use of expander graphs. If we additionally allow the algorithm to use randomness, however, we can construct a simple and concretely efficient algorithm that obliviously sorts n elements with only $O(n \log n)$ number of memory reads and writes, assuming that the client has at least super-logarithmic amount of local space.

At a high level, we will following the following blueprint for constructing an oblivious sort.

Given a sequence A:

1) Apply an **oblivious random permutation** (ORP) on A to get $A' \leftarrow \mathrm{ORP}(A)$.
2) Sort A' using any (potentially non-oblivious) comparison-based sorting algorithm, e.g., QuickSort.

Specifically, an oblivious random permutation (ORP) randomly permutes an input array, such that the access patterns do not reveal the permutation that has been applied.

Non-Oblivious Shuffle. The Fisher–Yates shuffle takes linear time to randomly permute n elements, but it is not oblivious. It works as follows: Given $A := [A_0, \cdots, A_{n-1}]$,

- for $i \in [0, n-1)$:
  - $j \xleftarrow{\$} [i, n)$;
  - Swap(A_i, A_j);

Apparently, one can exactly learn the random permutation applied by observing the Swap instructions.

Remark 3. We stress that in the second step of this "ORP and then sort" framework, it is important that the sorting algorithm be comparison-based. For example, we cannot use counting-based sort (where we just throw elements into bins based on their keys), because even after the sequence is randomly permuted, applying counting sort would reveal the frequency of elements in each bin, which is exactly what we want to hide. In contrast, a comparison-based sorting will only reveal the relative order between elements, which is fine after we randomly permuted the positions of elements.

Given the above blueprint, it suffices to construct an ORP that incurs $O(n \log n)$ memory reads and writes for permuting n elements.

Constructing an ORP. We present the Bucket Oblivious Sort algorithm by Asharov et al. [30]. We will first divide the input array into buckets of super-logarithmic size. Each element is then assigned a random label that denotes the destination bucket the element wants to go to. We then route the elements over a butterfly network of super-logarithmically sized buckets till all elements eventually land in their destined bucket. Finally, we randomly permute the elements inside each bucket. More specifically, the ORP algorithm works as follows.

- (Set-up:) Let the bucket size Z be super-logarithmic in some security parameter λ. Let $2n/Z$ be the number of buckets. Each bucket contains $Z/2$ *real* elements from the input sequence and $Z/2$ filler elements.
- Each element is randomly assigned a label of $\log B$ bits representing the destination bucket the element wants to go to.
- The elements are routed to their respective destinations through a butterfly network, which contains $\log(2n/Z)$ layers that are connected through MERGESPLIT operators described below.
- The MERGESPLIT operator takes two buckets from the i^{th} level as inputs and reorganizes their elements into two buckets for the $(i+1)^{\text{th}}$ layer, according to the $(i+1)^{\text{th}}$ most significant bit of the elements' keys. For example, an element labeled 01001 will go left at the first layer, go right at the second layer, go left at the third layer, and so on. In other words, at the i^{th} level, every $2i$ consecutive buckets are *semi-sorted* by the most significant i bits of the keys.
- Finally, apply an oblivious random permutation to permute the elements within each bucket, and remove all the fillers. See below for more details.

More details on the final step. To obtain a random permutation from our random bucket assignments, we obliviously and randomly permute the real elements within each bucket in the final layer. Further, we remove the filler elements from each bucket in the final layer. The bucket-wise oblivious random

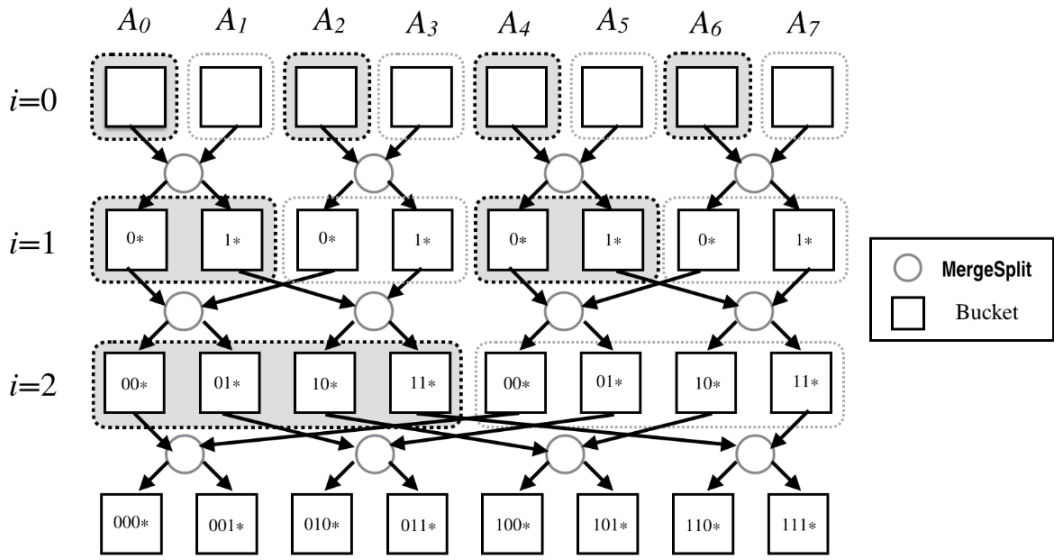


Fig. 4: Illustration of the MERGESPLIT process used in Bucket Oblivious Sort to randomly permute elements without leaking their original positions. Elements are routed through a butterfly network consisting of $\log(2n/Z)$ layers. At each layer, two buckets are merged and then split into two new buckets based on a specific bit of the element's random label. By using random destination labels, the elements are naturally balanced across the shards, ensuring no bucket overflows (except with negligible probability) [30].

permutation can be accomplished by assigning a random $\log n$-bit label to each real element in the bucket, and then calling bitonic sort to sort the real elements by their labels, while moving all filler to the end of the bucket (if two elements are found to have duplicate labels, we can repeat the process). The fillers can be removed by making a pass over the final-layer buckets, writing down only the real elements. During this process, the amount of filler in each final-layer bucket is revealed — but this information is safe to reveal since it is simulatable without knowledge of the contents of the input array to be sorted.

Efficiency analysis. If the client has a storage size no less than $2Z$, then they can perform MERGESPLIT and randomly permute the buckets at the final layer locally. In each of the $\log \frac{2n}{Z} < \log n$ layers, the $2n$-sized sequence (containing n real elements and n dummy elements) is read and written once. So ORP incurs $O(n \log n)$ memory reads and writes. Since the non-oblivious comparison-based sort also incurs $O(n \log n)$ cost, the resulting Bucket Oblivious Sort algorithm can be accomplished with $O(n \log n)$ memory reads and writes.

Success probability. The above algorithm is guaranteed to output a correct, randomly permuted version of the input sequence if none of the buckets ever overflows (i.e., receives more than Z real elements) during the routing.

Lemma III.1. *Overflow happens with at most* $\epsilon(n, Z) = 2n/Z \cdot \log(2n/Z) \cdot e^{-Z/6}$ *probability.*

Proof. Consider an arbitrary bucket B_i at level i. The bucket is associated with some i-bit prefix, and it receives real elements from 2^i initial buckets in the starting layer. Any element in these 2^i initial buckets whose label has a matching i-bit prefix is destined to go to this bucket B_i. The expected number of real elements destined for B_i is therefore $2^i \cdot \frac{Z}{2} \cdot \frac{1}{2^i} = \frac{Z}{2}$.

By a standard application of the Chernoff bound, bucket B_i overflows with at most $e^{-Z/6}$ probability. Hence, taking a union bound over all levels and all buckets, we get that except with $\epsilon(n, Z) := 2n/Z \cdot \log(2n/Z) \cdot e^{-Z/6}$ probability, no bucket ever overflows. □

Lemma III.1 shows that it suffices to set Z to be any super-logarithmic function (in the security parameter) for the failure probability to be negligibly small in the security parameter.

Additional improvements. The above efficiency analysis assumes that the client has at least super-logarithmic amount of local space. Ramachandran and Shi [31] showed how to get optimality when the client has constant space, by changing the 2-way butterfly network to a $\log n$-way butterfly network, and using AKS to perform each $\log n$-way MERGESPLT operation. Gu et al. [32] further improved the algorithm to achieve concrete efficiency while retaining asymptotical optimality (under constant client space).

B. Building Block: Oblivious Hash Table for Non-Recurrent Requests

We consider a standard static hash table abstraction.

Definition 1 (Static Hash Table)**.** Consider a data array $A = \{(k_i, v_i)|\perp\}_{i\in[n]}$, i.e., each element is either a *real* key-value pair denoted (k_i, v_i) or a *filler* denoted $\perp$. It is guaranteed that all real elements have distinct keys. An oblivious hash table contains the following operations:

- Build(A): Given a data array A, output a data structure.
- Lookup(k): Given a key k, output v. If $k = \perp$ or $k \notin A$, then output $v = \perp$; otherwise the output v satisfies $(k, v) \in A$.

The hash table we will construct satisfies obliviousness only if the same real key is never looked up twice. It turns out that such a hash table is sufficient for constructing hierarchical ORAM. We now formally define this non-recurrent requirement. A sequence of Lookup operations denoted op is said to be *non-recurrent* if every real key requested is distinct. However, the sequence op is allowed to make filler requests $k = \perp$ multiple times.

Obliviousness for non-recurrent lookups. Obliviousness requires the following: $\forall A_0, op_0, A_1, op_1$ s.t. $|A_0| = |A_1|, |op_0| = |op_1|$ and both op_0 and op_1 satisfy the non-recurrent constraint, then it holds that

$$\text{AccessPattern}(A_0, op_0) \approx \text{AccessPattern}(A_1, op_1)$$

, where AccessPattern(A, op) denotes the access patterns emitted by the oblivious hash table when making a single call to Build(A) followed by a sequence of Lookup operations denoted op, and $\approx$ means computational indistinguishability. More intuitively, we want that for any two data arrays and operation sequences of the same length, the access patterns in oblivious hash tables are indistinguishable as long as the same key is never looked up twice.

Construction. Now, let's introduce how to build an oblivious hash table. We use the balls and bins hashing with n elements (balls) and n/Z bins[2] where $Z = \omega(\log N)$. The expected number of elements in any fixed bin is Z. With a Chernoff bound, the probability of any bin having more than $2Z$ elements is negligible in N. Thus, we set the bin size to be $2Z$. One way to randomly assign the element (k, v) to a bin is to use a pseudorandom function $PRF_{sk}(k)$, where the sk is the secret key held by the ORAM client and the PRF output range is $[N/Z]$, indicating the desired bin index.

So now, we want to *obliviously* throw the balls into the bins without revealing which bin each ball goes to. At the end, each bin is padded with filler elements to the maximum capacity. To achieve this, we may use the "oblivious random bin assignment" construction — see Figure 4 of Section III-A[3].

To implement Lookup(k), if k is a real key, the client simply reads the entire bin indexed $PRF_{sk}(k)$. If $k = \perp$, then the client reads a random bin.

Analysis. Building an oblivious hash table calls the "oblivious random bin assignment" algorithm of Section III-A, which incurs $\mathcal{O}(n \log n)$ cost. For every lookup, the cost is the bin size, which is $Z = \omega(\log N)$.

[2]For simplicity, we assume that n is divisible by Z. For our final ORAM construction, we can assume that RAM's space N is at least the (statistical) security parameter, so we set the bucket size Z to be dependent on N rather than the security parameter.

[3]The original work of Goldreich and Ostrovsky [1] uses oblivious sorting to accomplish this, but we use oblivious random bin assignment for simplicity and "self-containedness."

C. Hierarchical ORAM

So now with this oblivious hash table, we can construct a Hierarchical ORAM. The hierachical ORAM essentially uses Bentley and Saxe's hierarchical data structure idea [9] that converts a static data structure (in our case, an oblivious hash table) into a dynamic one.

In Hierarchical ORAM, we have a hierarchy of exponentially growing oblivious hash tables as shown in Figure 5. Each level of the hierarchy has a different capacity, and the capacity grows exponentially with the level. The levels are indexed $0, 1, \ldots, L$, and the i-th level's capacity is 2^i. Without loss of generality, we shall assume that N is a power of 2, and L is chosen such that $2^L = N$. Additionally, every level has a label, either **empty** or **full**. "Empty" means that this level is currently not in use, and "full" means that this level is currently in use. Whether each level is empty or full only depends on the time step (i.e., how many operations have been performed so far), so this information is public and is known by the adversary.

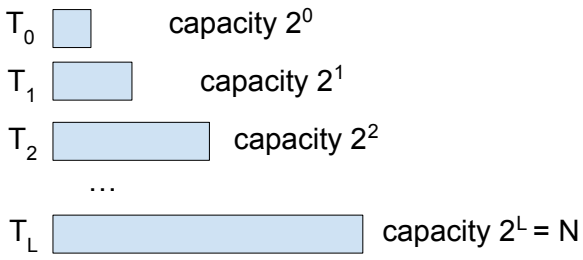


Fig. 5: This framework organizes memory into a series of exponentially growing oblivious hash tables, where each level i has a capacity of 2^i. To ensure security, lookups proceed from the smallest to largest level; once the target element is found, "fake" lookups are performed on all remaining levels to hide the location of the real data. After an access, elements are relocated and levels are periodically rebuilt using oblivious sorting to prevent access pattern leakage.

Recall that we want to implement a memory abstraction with the address space $1 \ldots N$. Initially, we may assume that all memory cells are empty and contain the symbol $\perp$. Therefore, initially all the ORAM levels are marked empty. However, later, as we perform read and write operations, at any point of time, some levels might be full and others are empty.

Now, suppose a sequence of operations arrive. There are two types of operations, `Read(addr)` and `Write(addr, data)`.

Intuition. Before diving into the detailed algorithm, let's first introduce the high-level idea of how to perform these operations.

If we want to look up some element, we will start from the smallest level. Every level is either empty or full. If it is full, look up that hash table for the desired element. There are two outcomes: either we find the element, or we don't find the element. If we have found the element, for all future levels, we just perform a fake lookup by requesting a filler element

Algorithm 3: HIERARCHICAL ORAM

Input: addr, an address to lookup in the hash tables; op, the operation write or read; data, the data to write.

Output: $data^*$, the lookup result, if op $=$ read.

found $\leftarrow$ $false$;
for $l \leftarrow 0$ **to** L **do**
  if *not* found **then**
    fetched $\leftarrow T_l$.Lookup(addr);
    if fetched $\neq \perp$: found $\leftarrow$ $true$;
     data* $\leftarrow$ fetched;
  else
    $T_l.lookup(\perp)$
if op $=$ read let $T^\phi \leftarrow \{(\texttt{addr}, \texttt{data}^*)\}$; **else if** op $=$ write let $T^\phi \leftarrow \{(\texttt{addr}, \texttt{data})\}$;
Let l be the smallest level such that T_l is empty, if all levels are full, let $l = L$;
Let $S \leftarrow T^\phi \cup T_0 \cup \cdots \cup T_{l-1}$; and if all levels are full, let $S \leftarrow S \cup T_L$;
T_l.Build(SuppressDup(S));
Mark T_l as full, mark $T_0, \cdots, T_{l-1}$ as empty;
if op $=$ read: **return** data*;

$\perp$. Similar to the tree-based ORAM, we will also need to relocate the retrieved element. So in Hierarchical ORAM, we will put the retrieved element to the smallest level if this level is empty. If this level is not empty, we will need to merge the retrieved element with the elements in this level and a merged hash table of size 2 is copied to the next level if it is empty. However, if the next level is not empty, then it causes a cascading merge until we find a level that is empty or we reach the largest level.

Algorithm. The detailed algorithm is shown in Algorithm 3. We use $T_0, \ldots T_L$ to denote the oblivious hash table of the $L+1$ levels. The algorithm starts by initiating a binary variable found to false. Then, for each level, we check if the element is in the hash table. If we find the element, we set found to true and store the element in data*. If the element is already found, we just look up a filler for all future levels — importantly, *this guarantees that the non-recurrent constraint is satisfied for every oblivious hash table*. After we go through all levels, the fetched element is in data*.

If the operation op is read, we will need to write back the retrieved element $T^\phi = \{(\texttt{addr}, \texttt{data}^*)\}$. Otherwise, if the operation op is write, we will need to write back the new element $T^\phi = \{(\texttt{addr}, \texttt{data})\}$. Let l be the smallest level such that T_l is empty, if all levels are full, let $l = L$. We want to merge all levels $0 \ldots l-1$ along with the new element into level l (or merge all levels $0 \ldots L$ along with the new element into the largest level L). Then, we obliviously build the hash table at level l with the input set S. After the build step, we mark the level l as full and the levels 0 to $l-1$ as empty.

Note that if the oblivious hash table doesn't remove the element we just looked up, there might be duplicate elements in the set S and we will need to suppress the duplicate elements. The elements in the smaller level is always fresher than the elements in the larger level, so we can just keep the fresher element and remove the older one. We can use the oblivious sorting to do the duplicate suppression, denoted SuppressDup in Algorithm 3. Specifically, we obliviously sort the elements by key and by freshness and remove the duplicate elements using a linear scan.

Analysis. Now, let's analyze the overhead of Hierarchical ORAM. For each request, there are two sources of the overhead: the lookup phase, and the rebuild phase. The lookup cost is $L \times Z$, the number of levels L times the lookup cost of at each level, which is at most the bin size Z. Recall we can set $Z = \log N \cdot \alpha(N)$, where $\alpha(N)$ is an arbitrarily small super-constant function. Thus, the lookup cost is $\mathcal{O}(\log^2 N \cdot \alpha(N))$.

For rebuild cost, a level of size 2^i is rebuilt every 2^i steps. So to rebuild a level of size 2^i, the cost is $2^i \cdot \log 2^i$ (see the analysis of the oblivious hash table). By amortizing the cost to each step, every level has a cost $\leq \log N$. Thus, the rebuild cost is $\mathcal{O}(\log^2 N)$.

Summarizing the above, each memory request has cost $\mathcal{O}(\log^2 N \cdot \alpha(N))$ where $\alpha(N)$ may be an arbitrarily small super-constant function. Note that the original Goldreich-Ostrovsky construction [1], [2] has a cost of $\mathcal{O}(\log^3 N)$, because they throw N balls into N bins, instead of N/Z bins.

Remark 4. Bentley and Saxe's hierarchical data structure idea [9] is a well-known and widely used technique for converting static data structures into dynamic ones, e.g., see lecture notes on amortized algorithms [33]. There are also various applications in the cryptography literature besides ORAM, e.g., to obtain dynamic versions of proofs of retrievability [34], searchable encryption [35], and private information retrieval [36].

D. Optimal Oblivious RAM

The above hierarchical ORAM has $O(\log^2 N \cdot \alpha(N))$ overhead, where one $\log N$ comes from the number of levels in the hierarchy, and another $\log N$ comes from the size of the hash bin for the read phase, and from the oblivious sort for the rebuild phase. It turns out that the latter $\log N$ can be removed with more sophisticated techniques. First, to remove the extra $\log N$ factor in the read phase, one can use an oblivious Cuckoo hashing scheme rather than balls-into-bins hashing [6], [16], [17], [37]. To remove the extra $\log N$ factor in the rebuild phase, one can accomplish the hash table rebuild relying on oblivious compaction (i.e., sorting elements with 1-bit keys) rather than oblivious sort (of general keys) [16]. Moreover, oblivious compaction can be accomplished in cost that is linear in the size of the input array [16]. Putting together all these techniques, one can get a computationally-secure ORAM with only $O(\log N)$ overhead [16], which is asymptotically optimal by the well-known ORAM lower bounds [1], [15]; see also Section IV. Because achieving this optimal bound requires additional technical machinery, we refer the reader to Asharov et al. [16] for further details.

IV. ORAM Lower Bound

In this section, we will prove that any Oblivious RAM (ORAM) scheme must suffer from logarithmic overhead. We will show two proofs. The first proof in Section IV-A was described in Goldreich and Ostrovsky's original paper on ORAM [1]. Their lower bound has two restrictions: 1) it works only for *statistically* secure ORAMs and 2) it assumes that the ORAM is in the *balls-and-bins* model, i.e., the scheme does not perform any encoding on the payload strings stored in memory. We will give a simplified proof for perfectly secure ORAMs, i.e., an unbounded adversary has zero probability of success. Years later, in 2018, Larsen and Nielsen [15] proved a new lower bound removing both restrictions of the Goldreich and Ostrovsky proof. Interestingly, their proof uses techniques from the data structure lower bound literature. We will also cover Larsen and Nielsen's lower bound in Section IV-B.

A. Goldreich and Ostrovsky's Lower Bound

Theorem IV.1 (Goldreich-Ostrovsky ORAM lower bound)**.** *Consider any perfectly secure ORAM scheme in the balls-and-bins model such that the memory is initialized with n words, and the client has m space. Then, any logical request sequence of length t must incur $\max(n, \Omega(t \log_m n))$ total cost. Further, the lower bound works even for read-only requests.*

Proof. Consider the following game. Initially, there are n balls, and ball i is stored in cell i of the memory. There is a sequence of t logical requests, to read the balls indexed $i_1, \ldots, i_t$ respectively. A player can hold up to m balls in her hand, and initially, her hand is empty. In every time step indexed $1, 2, \ldots, q$, she can visit a memory cell of her choice and take one of the following hidden actions:

1) Take a ball from the memory cell and put it in her hand;
2) Place a ball from her hand to the memory cell (if it is currently empty);
3) Do nothing.

The player's action sequence can satisfy the request sequence, iff there is a subsequence $1 \le j_1 \le j_2 \le \ldots \le j_t \le q$, such that for all $k \in [t]$, the ball indexed i_k is in the player's hands at the end of time step j_k.

Suppose that an adversary can observe which memory cell the player visits in every time step, but cannot observe which hidden action the player takes. Similarly, the adversary cannot observe which balls are stored in the memory cells or the player's hands. Now, the player's job is to satisfy the logical request sequence $i_1, \ldots, i_t$ without revealing any information about the logical request sequence. We assume that the adversary knows the parameters n, t, m.

First, it is easy to see that $q \ge m$. To see this, consider a logical request sequence of length 1. To satisfy the request, if there is some memory cell the player does not visit, it directly reveals that the request is not for that index. In the remainder of the proof, we focus on proving $q \ge \Omega(t \log_m n)$.

Consider a fixed sequence of memory cells visited $(v_1, \ldots, v_q)$ that happens with non-zero probability when the logical request sequence is $(1, 1, \ldots, 1)$. Because of the privacy requirement, $(v_1, \ldots, v_q)$ must be able to satisfy any logical request sequence of length t, and the total number of logical request sequences of length t is n^t.

Now, how many logical request sequences can $(v_1, \ldots, v_q)$ satisfy? When we fix the physical access sequence $(v_1, \ldots, v_q)$, in each of the q time steps, the player can choose one of at most $m+2$ hidden actions. Specifically, if the player chooses to place a ball, the ball can be one of the (up to) m balls in her hand. Therefore, fixing $(v_1, \ldots, v_q)$, there are $(m+2)^q$ possible action sequences. Further, when we fix the sequence of memory cells visited $(v_1, \ldots, v_q)$ as well as the sequence of hidden actions, it can satisfy at most $\binom{q}{t} \cdot m^t$ logical requests, where $\binom{q}{t}$ is the number of ways to choose t out of the q time steps, and for each of the t chosen time steps, the player can choose one out of up to m balls in her hand to satisfy the next request.

Summarizing the above, we have that

$$\binom{q}{t} \cdot (m+2)^q \ge n^t.$$

Using the fact that $\binom{q}{t} \le \left(\frac{eq}{t}\right)^t$, we have

$$\left(\frac{eq}{t}\right)^t \cdot (m+2)^q \ge n^t.$$

Therefore,

$$q \log(m+2) \ge t(\log n - \log(eq/t)).$$

If $q/t > \sqrt{n}$, then $q > t \log n$ trivially follows. Therefore, we may assume that $q/t \le \sqrt{n}$. In this case, we have that $q \ge \Omega(t \log n / \log(m+2))$ which gives the desired bound. □

In the above proof, the game setup where the player can only grab and place balls means that this proof works only for ORAM schemes in the balls-and-bins model. Moreover, the counting-based argument implicitly assumes that the scheme is perfectly secure. With some more work, it is possible to extend the above proof to work for statistical (rather than perfect) security.

B. Larsen and Nielsen's Lower Bound

Goldreich and Ostrovsky's lower bound suffers from two restrictions: 1) the ORAM scheme must be statistically secure; and 2) the ORAM scheme must follow the balls-and-bins model, i.e., the algorithm cannot perform any encoding of the payload strings.

We now show a more recent lower bound proof by Larsen and Nielsen [15] which removes these two restrictions. Their proof uses elegant techniques from the data structure lower bound literature, called the "information transfer" technique.

We want to realize a memory abstraction that supports two types of operations, `read(addr)` and `write(addr, data)`. For simplicity, we assume that the ORAM client has $O(1)$ space. Imagine a physical memory array that contains *cells* and each cell stores one word. Our ORAM algorithm will read/write some of the cells upon receiving a request.

In the proof, we will consider a scenario with $2T$ requests, and imagine that $T = \Theta(N)$ where N denotes the memory size. We will show that the following canonical request sequence of length $2T$ must make many cell probes.

$$op^* := \texttt{write}(0,0), \texttt{read}(0), \texttt{write}(0,0), \texttt{read}(0), \ldots$$

We will count the cell probes in a clever way, by assigning the cell probes to nodes in a binary tree (see Figure 6). Imagine a binary tree with T leaf nodes. Each leaf node i is associated with two operations $\texttt{write}(\texttt{addr}_i, \texttt{data}_i)$ and $\texttt{read}(\texttt{addr}'_i)$. Imagine that these operations occur in chronological order from left to right. Given a leaf node i, to satisfy the requests associated with it, we need to make some cell probes. We will charge these cell probes to ancestors of the leaf node i (including i itself) in the following way. Suppose a cell probe visits some cell that was last written to during the requests associated with some leaf node $j \le i$. Then, this cell probe is charged to the lowest common ancestor of i and j. Therefore, once we complete the entire request sequence on the leaf nodes, we can assign all cell probes to nodes in the binary tree. To get the total number of cell probes, we can sum up the cell probes assigned to each node of the tree.

Let $P_u(op)$ denote the expected number of cell probes assigned to some node u after executing the request sequence op. We want to show that $P_u(op^*)$ is large for any u. To do so, we will need to use the security requirement of ORAM, which says that the access patterns of any two request sequences of the same length must be computationally indistinguishable. Now, suppose that an adversary observes the cell probes for each request, it can then assign these probes to nodes in the binary tree in polynomnial time. Therefore, the security requirement of ORAM implies that for any request sequence op of the same length $2T$, $|P_u(op) - P_u(op^*)| \le \mathsf{negl}(n)$, and this must hold for any node u. For the rest of the proof, we will simply assume $P_u(op^*) = P_u(op)$ for any request sequence op of the same length — it is not hard to modify the argument to account for the negligibly small difference.

Now, to lower bound $P_u(op^*)$ for each u, we will come up with the *worst-case request sequence* for u, since the expected number of probes assigned to u under op^* is the same as under any request sequence op' of the same length. We use d to denote the height of the node u where the leaf is at height 0 and the root is at height $\log_2 T$. We use "Epoch L" to refer to the requests associated with leaves in the left subtree of u, and we use "Epoch R" to refer to the requests associated with leaves in the right subtree of u. See Figure 7 for an illustration.

Consider the following request sequence:

- *Before Epoch L*:

  $$\texttt{write}(0,0), \texttt{read}(0),\ \texttt{write}(0,0), \texttt{read}(0),\ \ldots,\ \texttt{write}(0,0), \texttt{read}(0)$$

- *Epoch L*:

  $$\texttt{write}(1, r_1), \texttt{read}(0),\ \texttt{write}(2, r_2), \texttt{read}(0),\ \ldots,\ \texttt{write}(2^d, r_{2^d}), \texttt{read}(0)$$

  where $r_1, r_2, \ldots, r_{2^d}$ are randomly and independently sampled words.

- *Epoch R*:

  $$\texttt{write}(0,0), \texttt{read}(1),\ \texttt{write}(0,0), \texttt{read}(2),\ \ldots,\ \texttt{write}(0,0), \texttt{read}(2^d)$$

- *After Epoch R*:

  $$\texttt{write}(0,0), \texttt{read}(0),\ \texttt{write}(0,0), \texttt{read}(0),\ \ldots$$

Informal intuition. The intuition is that to satisfy the read requests in Epoch R, we will need to transfer sufficient information from the write requests in Epoch L. Ignoring the $O(1)$ client space, information can only be transferred from the write requests in Epoch L to the read requests in Epoch R when Epoch R probes memory cells last written to during in Epoch L. Since the data strings written during Epoch L are all randomly chosen, roughly speaking, we need to transfer 2^d words of information from Epoch L to Epoch R, and thus the expected number of probes assigned to node u needs to be at least 2^d.

Using the same argument, we can find the worst-case request sequence for any node u, and argue that the expected number of probes assigned to u must be proportional to the subtree size of u. Thus, when we sum up all nodes u, we get that the total number of probes in the tree is at least $\Omega(T \log T)$. Amortized to $2T$ requests, it means that the average cost of each request is at least $\Omega(\log T) = \Omega(\log N)$.

Formal encoding argument. The above informal intuition can be formalized through an encoding-based argument. Specifically, we can prove that if there exists an ORAM scheme with $o(\log N)$ amortized cost per request, then we can leverage the ORAM scheme to compress a random string which is known to be impossible due to Shannon's fundamental theory on information theory. We defer the details of this encoding argument to the paper by Larsen and Nielsen [15], or Shi's lecture notes [38].

V. Applications of ORAM

Three broad classes of applications. Oblivious RAM has three broad classes of usage scenarios:

1) **Storage outsourcing.** In storage outsourcing, a weak client such as a laptop or mobile device outsources data its storage (e.g., file system or database) to an untrusted server. The client may need to query the remote data, e.g., selectively downloading some files or making a SQL database query. The client may also perform some local computation on the data, and it queries the server for the relevant data blocks as needed during the computation. Traditional storage outsourcing services include Dropbox, Google Drive, and others. A more modern example is blockchains' RPC services (e.g., Infura, Alchemy, and QuickNode). Specifically, light-weight blockchain clients often do not have enough space or bandwidth resources to participate in the full consensus protocol and store the entire blockchain. As a result, they fetch blockchain data as needed by making RPC calls to a server that runs the full consensus protocol and stores the entire blockchain log and states. The blockchain community has become increasingly concerned that queries to such RPC services leak highly sensitive information about the users' identities, trading intentions, as well as trading

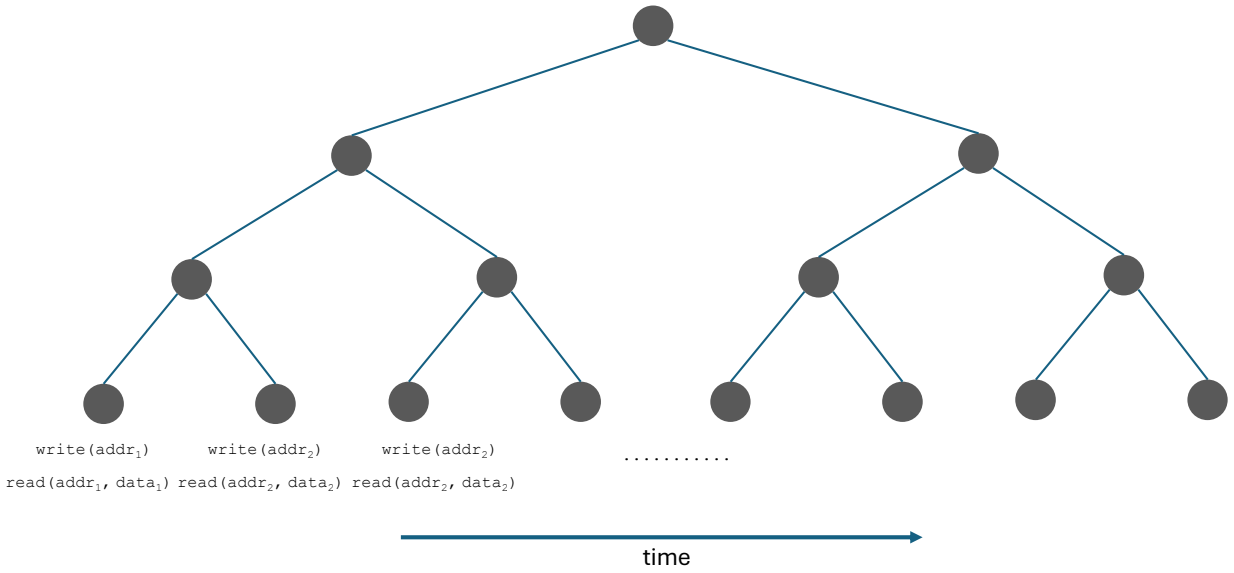


Fig. 6: Illustration of the tree-based model used to prove the $\Omega(\log N)$ lower bound for ORAM efficiency. It aids in counting "cell probes" by dividing the access sequence into distinct time intervals, such as Epoch L and Epoch R. By analyzing the information transfer required between these epochs——where random data strings written in one epoch must be retrieved in another——the proof demonstrates that the expected number of memory probes must be proportional to the subtree size, confirming that the $O(\log N)$ overhead is an inherent cost of memory obliviousness.

partners [39]. TaoStore [40] addressed oblivious storage under asynchronous and concurrent access patterns, which arise in distributed cloud settings.

2) **TEE and confidential cloud computation.** Trusted hardware, also known as TEE, short for Trusted Execution Environment, supports computation on normally encrypted data by creating a tamper-proof enclave — inside the enclave, data is decrypted and computation takes place. Software inside the TEE is trusted, and all software, including the operating system, that is outside the TEE is untrusted. The part of memory assigned to the enclave, termed enclave memory, is protected from reads and writes by the operating system, and can only be accessed by enclave software. Memory outside of the enclave memory region can be read and written by the operating system, and hence is untrusted. Enclave software can access untrusted memory, but the access is seen by the untrusted operating system.
   Many confidential computing applications require accesses to a large database. The database may not fully fit in the enclave's protected memory region, and is often stored in encrypted form in unprotected external memory or on disk. Accesses to such (encrypted) data stored in unprotected storage are therefore seen by the untrusted operating system. ORAM allows us to protect against such access pattern leakage in trusted hardware use cases. The ORAM position map, stash and ORAM state reside in enclave memory. The ORAM tree covers the database, which is stored outside of the enclave in untrusted memory. The ORAM client (i.e., algorithm) runs inside of the enclave, handling logical accesses and converting them to (many) physical accesses. Therefore, the untrusted OS or network adversary is blocked from seeing the pattern of accesses to the database.
   Industry leaders such as Signal [13], Meta [14], and Apple [41] have either deployed or committed to deploy trusted hardware to enable confidential computation (including AI computation) in the cloud.

3) **Multi-party computation.** Instead of using trusted hardware to realize a secure computer, we can alternatively use 2-party or multi-party computation (2PC or MPC for short) [42], [43] to realize a cryptographically-secure computer, by leveraging a set of distributed players who interact with each other to perform the computation. MPC guarantees that besides the output, no party learns any additional information about the secret inputs. During the early days, the MPC literature focused on the circuit model of computation. In other words, to securely compute some program, one has to first convert the program to a circuit, and then invoke a circuit-based MPC protocol. Unfortunately, the program to circuit conversion generally incurs a linear blowup in cost — in particular, every memory access is translated into a linear-sized circuit that encodes the entire memory tape!
   To reduce this program-to-circuit representation overhead, more recent literature explored 2PC or MPC for Random Access Machines (RAMs), also called RAM-model 2PC or MPC [44]. Conceptually, a set of players each store a secret share of the memory tape as well as the inputs, and they interact through MPC to securely emulate each CPU step. To make this work, the addresses to read and write must be declassified to the players during each CPU step, so that they can read and write shares of the value at the intended address. Therefore, it is important

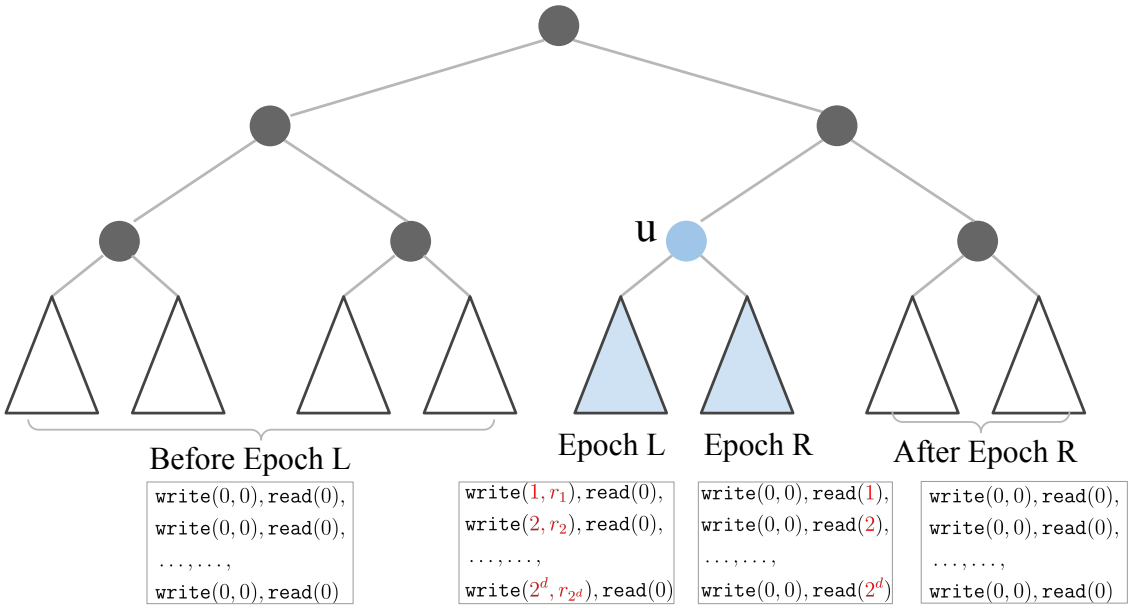


Fig. 7: Illustration of the proof of the efficiency lower bounds of ORAM using an "information transfer" technique, where a worst-case sequence of requests is used to measure the minimum number of physical memory "probes" required to maintain security. The proof relies on the fact that randomly and independently sampled data strings $r_1, \ldots, r_{2^d}$ must be handled such that their access patterns remain indistinguishable.

that we first compile the program to be evaluated (using the generic ORAM compiler) to an oblivious counterpart, to ensure that the memory addresses disclosed along the way do not leak the secret inputs. More recent work, called Garbled RAM [45]–[49], has also shown how to minimize the round complexity of RAM-model 2PC to just two rounds. Zahur et al. [50] revisited the classical square-root ORAM and showed that, with careful design, it achieves very efficient random access suitable for secure multiparty computation protocols.

Case study: Signal's private contact discovery. Signal is an encrypted messenger app with tens of millions of users. They want to support a private contact discovery application. In contact discovery, a user Alice sends her address book to Signal, and Signal will look up its user database stored in encrypted form on untrusted storage, and return Alice the information about her friends. The problem is that many users want to keep their address book private, and Signal wants to provide contact discovery without learning the users' contacts.

Signal solves this problem using trusted hardware enclaves and ORAM [13], as shown in Figure 8. Signal relies on commercial secure processors such as Intel's SGX to provide a tamper-proof hardware *enclave* on the server, also called a Trusted Execution Environment (TEE). The user encrypts its address book and sends it directly to the enclave. The address book is decrypted only inside the enclave, and using an ORAM algorithm, the enclave fetches the (encrypted) records corresponding to the user's friends from Signal's user database, and returns them to the user. Note that any data in transit or at rest is always encrypted, so the contents are protected from the server. Using the ORAM algorithm prevents the server from learning anything from the accesses of the user database. Because SGX enclaves leak information about enclave memory accesses at page-level granularity, there are additional complications in Signal's implementation (see [13]).

Besides Signal, Meta has also implemented ORAM [14] and is planning to launch a private cloud computing application. Ethereum is also planning to deploy infrastructure-level services that enable private reads to blockchain data, relying on ORAM and trusted hardware [39].

VI. Beyond ORAM

A. Oblivious Algorithms

ORAM provides a *generic* way to compile *any* program to an oblivious counterpart. This generality, however, often comes at a price. Specifically, when faced with a concrete computation task, directly using ORAM to compile a known insecure algorithm often leads to suboptimal performance. Notably, the $\Omega(\log N)$ lower bound should be viewed as the inherent cost when we insist on the generality. For specific computational tasks, this logarithmic lower bound may not be applicable.

In fact, when we construct ORAM schemes, we build upon some underlying oblivious primitives. We can often take advantage of these underlying oblivious primitives to construct customized oblivious algorithms tailored for the specific computational task of interest. We will give some examples below.

1) Oblivious Data Structures: Suppose we want to construct an oblivious binary search tree (BST). For simplicity, in this tutorial, we focus on a static BST that supports only lookups; but it is not hard to extend this construction to

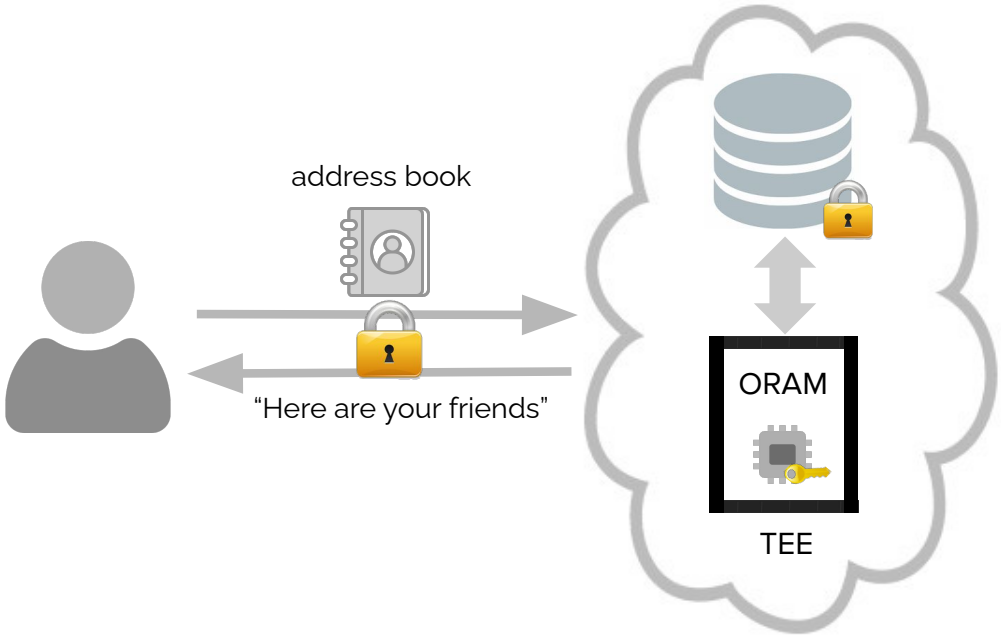


Fig. 8: Signal's private contact discovery app. The server employs a trusted hardware enclave also called a TEE. The client sends its encrypted address book to the server. The address book is decrypted inside the enclave, and using an ORAM, the relevant (encrypted) entries are fetched from unprotected storage without leaking the user's contacts.

additionally support insertions, deletions, as well as nearest-neighbor queries.

Naïve approach: $O(\log^3 N)$ cost per operation. The naïve approach is to use Path ORAM [11] to directly compile a non-oblivious BST, e.g., an AVL tree — henceforth we shall call the AVL tree the *logical* tree to distinguish it from the ORAM tree. The cost per operation for a non-oblivious AVL tree is $O(\log N)$ where N is the maximum number of elements in the data structure, and the Path ORAM compilation incurs $O(\log^2 N)$ multiplicative overhead. So the naïve approach would incur $O(\log^3 N)$ cost per lookup operation. Note that in an AVL tree, not all operations take the same amount of time. When we use ORAM to compile an AVL tree, we have to pad all operations to the same length to avoid leakage through the number of accesses.

If we directly use Path ORAM to compile the AVL logical tree, essentially we have to go through the entire ORAM recursion for reading every node in the logical tree. The following simple idea [51], [52] allows us to avoid this recursion overhead.

Improved construction: piggybacking the logical tree over the ORAM's recursion data structure. As shown in Figure 9, each element in the ORAM tree corresponds to a node in the logical tree. We will have each element additionally store the position identifiers of its two children. Moreover, the client always stores the position identifier of the logical root. When the client has fetched some node in the logical tree, it immediately knows the position identifiers of its two children, and hence there is no more need to do a recursive position map lookup to find out the next child node it wants to read. Using this idea, we need to visit only $O(\log N)$ ORAM tree paths for each lookup request. Therefore, the cost per **Lookup** is reduced to $O(\log^2 N)$.

Remark 5. In a non-oblivious AVL tree, the lookup algorithm stops whenever the desired key is found. However, in an oblivious BST, such early stopping would leak information about where the requested element is in the logical tree. Therefore, even after we have found the element, we still have to perform fake operations on the ORAM tree, such that all requests incur the same number of memory accesses.

Supporting updates. The above idea was first suggested by Gentry et al. [51] who considered a static BST. Subsequently, Wang et al. [52] showed that it is not hard to extend the idea to support dynamic insertions and deletions. Specifically, during an insertion or deletion operation, the logical AVL tree may perform rotation operations to balance the logical tree. Each rotation operation involves modifying the pointers on constant number of nodes in the logical AVL tree. In our case, during a rotation, we just have to additionally update the children position identifiers for nodes involved in the rotation. Again, we have to pad all requests to the same length to avoid leakage through early stopping. It is also not hard to support nearest neighbor queries using similar ideas.

Other data structures. We can extend this idea to get oblivious stacks and queues with $O(\log N)$ cost per operation. For priority queues, although this approach also applies, it yields suboptimal performance. Shi [53] showed a better way to construct an oblivious priority queue. Specifically, an oblivious priority queue can be constructed using a simple modification

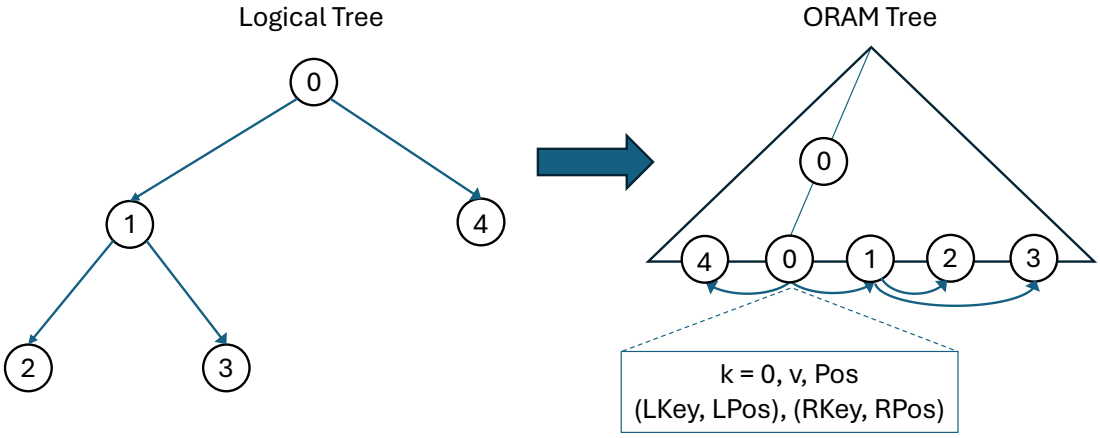


Fig. 9: Each element in the ORAM tree corresponds to a node in the logical tree. Each element stores the position identifiers of its two children. This eliminates the need to perform a recursive position map lookup for the next child node to read. Note that the element is actually stored somewhere along the path leading to the leaf even though we draw it at the leaf. $(\mathsf{LKey}, \mathsf{LPos})$ and $(\mathsf{RKey}, \mathsf{RPos})$ denote the keys and the position identifiers of the two children. Pos denotes the position identifier of the current element, which is needed for performing eviction on the ORAM tree.

to an ORAM tree without recursion, resulting in $O(\log N)$ cost per query — matching the cost of a non-oblivious binary heap for `DecreaseKey`, `ExtractMin` and `Insert` operations.

2) Applications of Oblivious Sorting: Recall that oblivious sorting is a primitive used in constructing the hierarchical ORAM. It is well-known that many applications can be built from sorting, e.g., list ranking, tree computations with Euler tour, tree contraction, graph connectivity, and minimum spanning forest [31]. For such applications, we can get efficient oblivious algorithms without going through the generic ORAM transformation. In fact, sorting is also a fundamental building block in the design of parallel algorithms. For this reason, many parallel algorithms can be made oblivious without the overhead of generic ORAM transformation — see also Section VI-B.

Oblivious sorting is used by Signal to shard the ORAM and spread the query load across multiple CPU cores — the sharding techniques here are closely related to the body of work on Oblivious Parallel RAM (OPRAM) from the theoretical literature [54], [55]. Imprecisely speaking, when a batch of queries arrive, we first use oblivious sorting to perform deduplication. After deduplication, all remaining requests are distinct, each wanting to go to a random shard, so the load between shards is naturally balanced. Once we obtain the answers to the (deduplicated) queries, we again use oblivious sorting to route the answers back to the requesters.

As ORAM and oblivious algorithms gain traction in large-scale applications, the community has also begun new efforts to develop an open-source Oblivious STL library [56], aiming to provide hand-optimized oblivious implementations of common utilities, data structures, and algorithmic building blocks such as those shipped with standard libraries such as C++'s STL or Rust's std. Last but not the least, several works have also explored lower bounds for oblivious data structures [57], [58] and oblivious sorting [29], [59].

B. Models of Computation

Depending on the application in which an oblivious algorithm is deployed, we use different computational models to capture the relevant performance metrics.

Word-RAM model. Literature on cloud computing [21], [60] often considers a client device (e.g., a laptop or mobile phone) with a moderately-sized private cache, whereas the MPC literature typically assumes the ORAM's client has an $O(1)$-sized private cache (since accesses to the private cache incur cost in the MPC circuitry too). However, the cloud outsourcing and MPC literatures share something in common: they both assume an abstract machine (commonly known as a *word RAM*) where the memory reads and writes happen at the granularity of words. A word-RAM is also the default, implicit model of computation adopted in standard undergraduate algorithms textbooks [4], [5]. This model captures a simple machine model where after carrying out the next CPU instruction, we read and write *one word* from the memory. Thus, in the word-RAM model, the *total computational work* and *number of memory accesses* are the same, and we typically do not distinguish them. The same metric is also referred to as the *bandwidth cost* in some works that focus on the cloud computing setting [21], [60], [61].

External-memory model. The aforementioned word-RAM model, however, is not the best fit for TEE-based oblivious computation. Instead, real-world TEE-based deployments of oblivious algorithms require algorithms optimized for the so-called external-memory model [62], where reads and writes to external memory happen at the granularity of pages (also called blocks) rather than words. Specifically, whenever the hardware enclave wants to read (encrypted) data that lives outside the enclave, data are transferred in and outside the enclave at a page granularity (typically 4KB), relying on a page swap operation. These page swaps are heavyweight since

they involve encryption/decryption, as well as potential context switches. Therefore, with TEEs, we should simultaneously optimize the *page swap cost* and the *computation cost* of the algorithm. This is exactly aligned with the external-memory model of algorithms design, first described by Aggarwal and Vitter [62].

As TEE-based oblivious computation gains traction in practice [13], [14], [39], it has become increasingly more important to optimize oblivious algorithms for the external-memory model of computation. Indeed, a more recent line of work has been dedicated to understanding what is the optimal performance of oblivious algorithms in the external-memory model [28], [31], [63]–[67].

Oblivious simulation on parallel architectures. Oblivious simulation of programs has also been explored in several parallel models of computation:

- **Parallel RAM.** Parallel RAM (PRAM) is a computational model meant to capture multi-core, shared-memory architectures. For a PRAM program, the primary performance metrics are *total work* and *depth*. Total work captures the program's running time if there were only one processor, and depth captures the program's running time if there were infinitely many processors. It has been shown that any PRAM algorithm can be compiled to a *statistically* oblivious PRAM with $O(\log^2 N)$ blowup in total work and $O(\log N \log\log N)$ blowup in depth, or compiled to a *computationally* oblivious PRAM with $O(\log^2 N/\log\log N)$ blowup in total work and $O(\log N)$ blowup in depth [55], [68]. Observe that the blowup in depth is smaller than the blowup in work — this is because the oblivious simulation itself exhibits some parallelism even if the original program is completely sequential.
- **Massively Parallel Computation.** Massively Parallel Computation (MPC[4]) [69] is a computational model meant to capture real-world cluster-based parallel architectures, commonly known as MapReduce or Spark. It has been show that any MPC program can be compiled into an oblivious MPC program with only *constant* blowup in round complexity [70].

Parallelism facilitates oblivious simulation. Sorting is a fundamental primitive in the design of parallel algorithms. It is known that if an algorithm has an efficient implementation in common parallel programming abstractions including MapReduce [37], [70], GraphLab [71], and fork-join [31], then we can obliviously implement the algorithm using oblivious sorting, without going through the generic ORAM compilation.

In other words, these programming abstractions capture large classes of computations that are obliviousness-friendly, i.e., they admit oblivious implementations that are fundamentally faster than generic ORAM compilation.

Other models that capture classes of obliviousness-friendly computation. Besides the aforementioned parallel programming abstractions, recent work has also shown other models that capture broad classes of computation that are obliviousness-friendly such as Oblivious Single Access Machine [72] and Oblivious Immutable Memory [73].

[4]In this paragraph, MPC refers to Massively Parallel Computation rather than Multi-Party Computation.

C. ORAM and Programming Languages

Implementing oblivious algorithms (or any cryptographic algorithm) can be error-prone in practice. For example, Tinoco et al. [66] observed some subtle obliviousness violations in earlier oblivious algorithm implementations [74]. Signal's ORAM implementation also has some subtle bugs that lead to slight information leakage, which they plan to amend in their next release. Therefore, an important question is whether we can develop formal verification techniques that programmatically guarantee the obliviousness of real-life implementations. Notably, Signal is planning on adopting formal verification techniques to mechanically prove the security of their ORAM implementation [75].

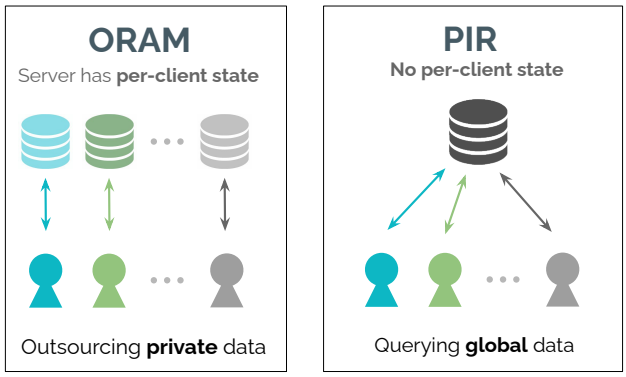
A memory-trace oblivious type system, initially proposed by Liu et al. [76], is one promising approach for achieving formal verification at compile time. Specifically, a memory-trace oblivious type system ensures that any well-typed program must satisfy obliviousness. Such a type system has also been applied to various architectures of oblivious computation, including multi-party computation [22] and ORAM-enhanced secure processors [77].

D. Comparison with Private Information Retrieval

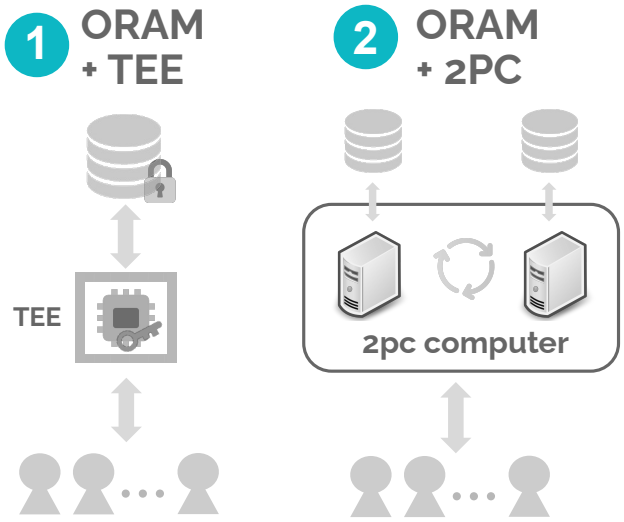
Oblivious RAM (ORAM) is reminiscient of another cryptographic primitive called Private Information Retrieval (PIR) [78]. Both ORAM and PIR aim to hide the access patterns to data. The main difference is that ORAM requires the server to store an ORAM'ed and encrypted copy of the dataset for every client, so it is a good fit for data outsourcing applications. In comparison, PIR is more suitable for private accesses to a global database, since the PIR server(s) need not keep per-client state. However, server-side computation in PIR is typically more expensive than ORAM. See also Figure 10a.

Interestingly, the real-life deployments of Signal and Ethereum's use cases actually desire PIR since they have a globally shared database. Why are they using ORAM? This is because they use a secure processor on the server which acts as a globaly trusted client for all users. This way, the server need not keep per-client state. Users send their encrypted queries to the secure processor on the server, and the secure processor performs the query using ORAM and returns the encrypted answer — see Figure 10b (left). Of course, the use of the secure processor requires trust in the hardware vendor. Therefore, these projects have it in their agenda to eventually replace the trusted hardware with cryptography. There are two approaches for achieving this:

1) replace the trusted hardware with a cryptographically-secure processor enabled by 2-party or multi-party computation, e.g., by employing an efficient Garbled RAM scheme [49] — see Figure 10b (right); or
2) employ a PIR scheme.



(a) ORAM vs. PIR.



(b) Using ORAM as a PIR through a secure computer enabled either by trusted hardware (also called TEE) or through cryptographic multi-party computation.

Fig. 10: Comparison of ORAM and PIR, two methods for protecting memory access patterns. ORAM requires a stateful client to constantly relocate data blocks to new random locations, achieving low communication overhead ($O(\log^2 N)$) suitable for large-scale databases. In contrast, PIR typically uses a stateless client and a static server, but requires the server to scan the entire database for every request, resulting in high computational costs ($O(N)$).

TABLE I: Performance of various approaches.

Scheme	TEE + Circuit ORAM	Garbled RAM	PIR (client-preprocessing)
Performance	20-50 μs $\sim$ 200K queries/sec	$\sim$ 30 ms online $\sim 300-600$ ms total	$\sim 10-20$ ms compute $\sim$ 90 ms total

The former approach requires a threshold-based trust model, whereas the latter approach can be built upon a single untrusted server. One main advantage of using a Garbled RAM is that it readily supports general-purpose computation and updates to the database are easy. In comparison, for PIR-based schemes to support general-purpose computation, we effectively need RAM-model Fully Homomorphic Encryption [79] which remains a theoretical concept and has a long way to go towards practicality. There are also some other known barriers for deploying PIR schemes at scale. Specifically, classical PIR schemes that do not employ preprocessing require linear server computation per query — even if we could parallelize the latency away, the computing bill would still be prohibitively expensive. For example, a few years back, Signal estimated that if they were to use a (then) state-of-the-art classical PIR scheme, it would have resulted in a very large annual computing bill. Although more recent literature has shown how to leverage client-side preprocessing to overcome the linear computation barrier and achieve practicality, the resulting schemes are not the best fit for scenarios with fast database updates, since the clients would have to be involved in the protocol for every update.

Concrete performance comparison. Table I provides a performance comparison of state-of-the-art implementations of the three approaches. The TEE+ORAM and Garbled RAM experiments are run on a machine with Intel Xeon Platinum 8352S CPU (8 cores at 2.2 GHz). The PIR experiment is performed on two AWS m5.8xlarge instances with 128GB of RAM.

The TEE+ORAM and Garbled RAM experiments are run on databases with 2^{30} entries. The TEE+ORAM assumes 8-byte keys and 56-byte values, the Garbled RAM stores 8-byte records. The Garbled RAM numbers are extrapolated from a smaller-scale experiment due to insufficient memory provisioning. The PIR experiment is on a 100GB database with 64-byte entries.

The TEE + Circuit ORAM benchmarks are obtained from Oblivious Labs' open-source implementation [80], the fastest implementation known to date. The Garbled RAM benchmarks are measured using PicoGRAM [49] using two variants of the scheme, one optimized for online response time, and the other optimized for the overall time (including offline and online costs). The PIR benchmarks are measured using Piano [81].

VII. Conclusion

Oblivious RAM has grown from a foundational cryptographic idea into a practical, deployed mechanism for eliminating memory-access leakage in untrusted environments. We

traced the evolution from the original hierarchical constructions, to statistically secure tree-based designs, to modern optimizations such as Path ORAM and Circuit ORAM that make ORAM feasible in real systems.

The binary-tree framework demonstrated that simple algorithms — mapping each block to a random path, relocating it on every access, and performing structured eviction — are enough to achieve strong obliviousness with predictable performance. Path ORAM and Circuit ORAM refined these ideas, reducing bucket sizes, improving eviction efficiency, and enabling deployment in settings ranging from secure processors to MPC protocols. The hierarchical framework, in parallel, showed how pseudorandomness and oblivious hashing can achieve optimal asymptotic bounds, filling in the theoretical landscape.

Today, ORAM sits at the center of several real-world applications: private contact discovery, confidential cloud services, and privacy-preserving machine learning pipelines. As systems increasingly rely on encrypted computation, ORAM provides a crucial guarantee by ensuring that even metadata such as access patterns cannot be exploited.

Looking forward, several challenges remain open. Reducing bandwidth overhead, improving locality, integrating ORAM with hardware accelerators, and formally verifying obliviousness properties are active areas of research. Emerging applications such as federated learning, and large-scale secure databases will continue to push ORAM toward more scalable, specialized designs.

References

[1] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, 1996.

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

[3] R. Pass and A. Shelat, "A course in cryptography," https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.

[5] J. Kleinberg and E. Tardos, *Algorithm Design*. Addison Wesley, 2006.

[6] T. H. Chan, Y. Guo, W. Lin, and E. Shi, "Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM," in *ASIACRYPT*, 2017, pp. 660–690.

[7] E. Boyle and M. Naor, "Is there an oblivious RAM lower bound?" in *ITCS*, 2016.

[8] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[9] J. L. Bentley and J. B. Saxe, "Decomposable searching problems I: static-to-dynamic transformation," *J. Algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

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

[11] 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 *CCS*, 2013.

[12] X. S. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *CCS*, 2015.

[13] "Technology deep dive: Building a faster oram layer for enclaves," https://signal.org/blog/building-faster-oram/.

[14] "Meta's oram implementation," https://github.com/facebook/oram.

[15] K. G. Larsen and J. B. Nielsen, "Yes, there is an oblivious ram lower bound!" in *CRYPTO*, 2018.

[16] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal Oblivious RAM," in *Advances in Cryptology - EUROCRYPT 2020*, 2020, to appear. See also: https://eprint.iacr.org/2018/892.

[17] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "PanORAMa: Oblivious RAM with Logarithmic Overhead," in *FOCS*, 2018.

[18] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," *J. ACM*, vol. 65, no. 4, pp. 18:1–18:26, 2018.

[19] M. Harchol-Balter, "Performance modeling and design of computer systems," Cambridge University Press, 2013.

[20] J. Hsu and P. Burke, "Behavior of tandem buffers with geometric input and markovian output," *IEEE Transactions on Communications*, vol. 24, no. 3, pp. 358–361, 1976.

[21] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *USENIX Security*, 2015.

[22] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A Programming Framework for Secure Computation," in *IEEE S & P*, 2015.

[23] K. Chung, Z. Liu, and R. Pass, "Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead," in *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 8874. Springer, 2014, pp. 62–81.

[24] A. Chakraborti, A. J. Aviv, S. G. Choi, T. Mayberry, D. S. Roche, and R. Sion, "rORAM: Efficient Range ORAM with $O(\log^2 N)$ Locality," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[25] 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 *PETS*, 2013.

[26] K. E. Batcher, "Sorting networks and their applications," in *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, 1968, pp. 307–314. [Online]. Available: https://doi.org/10.1145/1468075.1468121

[27] M. Ajtai, J. Komlós, and E. Szemerédi, "An O(n log n) sorting network," in *STOC*, 1983.

[28] W. Lin, E. Shi, and T. Xie, "Can we overcome the $n \log n$ barrier for oblivious sorting?" in *SODA*, 2019.

[29] A. Farhadi, M. Hajiaghayi, K. G. Larsen, and E. Shi, "Lower bounds for external memory integer sorting via network coding," in *STOC*, 2019.

[30] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket Oblivious Sort: An Extremely Simple Oblivious Sort," in *2020 Symposium on Simplicity in Algorithms (SOSA)*, 2020, pp. 8–14.

[31] V. Ramachandran and E. Shi, "Data oblivious algorithms for multicores," in *SPAA*, 2021.

[32] T. Gu, Y. Wang, A. Tinoco, B. Chen, K. Yi, and E. Shi, "Flexway o-sort: enclave-friendly and optimal oblivious sorting," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[33] "Lecture notes on amortized algorithms," https://www.cs.cmu.edu/~15451-s23/lectures/lec03-amortized.pdf.

[34] E. Shi, E. Stefanov, and C. Papamanthou, "Practical dynamic proofs of retrievability," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[35] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable symmetric encryption with small leakage," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[36] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 875–892. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/kogan

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

[38] "Lecture notes on oblivious ram and private information retrieval," https://www.cs.cmu.edu/~15893/syllabus.html.

[39] V. Buterin, "A maximally simple l1 privacy roadmap," https://ethereum-magicians.org/t/a-maximally-simple-l1-privacy-roadmap/23459.

[40] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *IEEE S & P*, 2016.

[41] "Private cloud compute: A new frontier for ai privacy in the cloud," https://security.apple.com/blog/private-cloud-compute/.

[42] A. C.-C. Yao, "Protocols for secure computations (extended abstract)," in *FOCS*, 1982.

[43] O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game," in *STOC*, 1987.

[44] 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*, 2012.

[45] S. Lu and R. Ostrovsky, "How to Garble RAM Programs," in *EUROCRYPT*, 2013.

[46] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs, "Garbled ram revisited," in *EUROCRYPT*, 2014, pp. 405–422.

[47] D. Heath, V. Kolesnikov, and R. Ostrovsky, "Epigram: Practical garbled RAM," in *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 13275. Springer, 2022, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-031-06944-4_1

[48] ——, "Tri-State Circuits: A Circuit Model that Captures RAM," in *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 128–160. [Online]. Available: https://doi.org/10.1007/978-3-031-38551-3_5

[49] T. Gu, A. Tinoco, S. H. G. Rajan, and E. Shi, "PicoGRAM: Practical Garbled RAM from Decisional Diffie-Hellman," in *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part VIII*, ser. Lecture Notes in Computer Science, vol. 16007. Springer, 2025, pp. 247–281.

[50] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting Square–Root ORAM: Efficient Random Access in Multi-Party Computation," in *IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 2016, pp. 21–40.

[51] C. Gentry, S. Halevi, M. Raykova, and D. Wichs, "Outsourcing Private RAM Computation," in *FOCS*, 2014.

[52] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious Data Structures," in *CCS*, 2014.

[53] E. Shi, "Path oblivious heap: Optimal and practical oblivious priority queue," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 842–858.

[54] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious Parallel RAM," in *Theory of Cryptography Conference (TCC)*, 2015.

[55] T. H. Chan and E. Shi, "Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs," in *Theory of Cryptography Conference, (TCC)*, 2017.

[56] "Open-source Oblivious STL library for Rust," https://github.com/obliviouslabs/rostl.

[57] R. Jacob, K. G. Larsen, and J. B. Nielsen, "Lower bounds for oblivious data structures," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '19. USA: Society for Industrial and Applied Mathematics, 2019, p. 2439–2447.

[58] K. G. Larsen, T. Malkin, O. Weinstein, and K. Yeo, "Lower bounds for oblivious near-neighbor search," in *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*. SIAM, 2020.

[59] T.-H. H. Chan, K.-M. Chung, B. M. Maggs, and E. Shi, "Foundations of differentially oblivious algorithms," in *SODA*, 2019.

[60] E. Stefanov and E. Shi, "ObliviStore: High Performance Oblivious Cloud Storage," in *IEEE S & P*, 2013.

[61] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[62] A. Aggarwal and S. Vitter, Jeffrey, "The Input/Output Complexity of Sorting and Related Problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[63] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Cache-oblivious and data-oblivious sorting and applications," in *SODA*, 2018.

[64] Z. Jafargholi, K. G. Larsen, and M. Simkin, "Optimal oblivious priority queues and offline oblivious RAM," Cryptology ePrint Archive, Report 2019/237, 2019, https://eprint.iacr.org/2019/237.

[65] I. Komargodski and W. Lin, "A logarithmic lower bound for oblivious RAM (for all parameters)," *SIAM J. Comput.*, vol. 54, no. 2, pp. 503–544, 2025.

[66] A. Tinoco, S. Gao, and E. Shi, "Enigmap: External-memory oblivious map for secure enclaves," Cryptology ePrint Archive, Paper 2022/1083, 2022, https://eprint.iacr.org/2022/1083. [Online]. Available: https://eprint.iacr.org/2022/1083

[67] G. Asharov, E. Eiluz, I. Komargodski, and W.-K. Lin, "MegaBlocks: Breaking the logarithmic i/o-overhead barrier for oblivious RAM," in *ACM CCS*, 2025.

[68] T. H. Chan, K. Chung, and E. Shi, "On the Depth of Oblivious Parallel RAM," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 10624. Springer, 2017, pp. 567–597.

[69] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '10. USA: Society for Industrial and Applied Mathematics, 2010, p. 938–948.

[70] T. H. Chan, K. Chung, W. Lin, and E. Shi, "MPC for MPC: Secure Computation on a Massively Parallel Computing Architecture," in *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, Seattle, Washington, USA, January 12-14, 2020*, ser. LIPIcs, vol. 151. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 75:1–75:52.

[71] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. USA: IEEE Computer Society, 2015, p. 377–394. [Online]. Available: https://doi.org/10.1109/SP.2015.30

[72] A. Appan, D. Heath, and L. Ren, "Oblivious single access machines - a new model for oblivious computation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 3080–3094.

[73] A. Appan and D. Heath, "Oblivious immutable memory," Cryptology ePrint Archive, Paper 2025/593, 2025. [Online]. Available: https://eprint.iacr.org/2025/593

[74] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy*, 2018.

[75] Private communication with Rolfe Schmidt.

[76] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *CSF '13*, 2013, pp. 51–65.

[77] C. Liu, M. Hicks, A. Harris, M. Tiwari, M. Maas, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *ASPLOS*, 2015.

[78] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1995, pp. 41–50.

[79] W.-K. Lin, E. Mook, and D. Wichs, "Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe," in *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, ser. STOC 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 595–608. [Online]. Available: https://doi.org/10.1145/3564246.3585175

[80] "Oblivious key-value store by oblivious labs," https://github.com/obliviouslabs/oram.

[81] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: Extremely simple, single-server pir with sublinear server computation," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 4296–4314.