Enclave-Based Oblivious RAM Using Intel's SGX

Maan Haj Rachid[1], Ryan Riley[2], and Qutaibah Malluhi[3]

[1]maan.rachid@scilifelab.se, Karolinska Institutet
[2]rileyrd@cmu.edu, Carnegie Mellon University in Qatar
[3]qmalluhi@qu.edu.qa, Qatar University

Abstract—Oblivious RAM (ORAM) schemes exist in order to protect the access pattern of data in a data store. Under an ORAM algorithm, a client accesses a data store in such a way that does not reveal which item it is interested in. This is typically accomplished by accessing multiple items each access and periodically reshuffling some, or all, of the data in the data-store. While many recent schemes make the ORAM computation complexity feasible, the performance of practical implementations is still largely limited by computational and storage limitations of the client as well as the bandwidth available between the client and the data store. In a cloud computing environment, where it is commonly assumed that the client is underpowered and you must pay by the gigabyte for data transfer, traditional ORAM methods are not optimal.

Intel's Software Guard Extensions (SGX) provide a new opportunity for ORAM implementations that can safely outsource the computational and bandwidth requirements along with the data itself, meaning that the client can be very limited and still attain high performance. In this work, we develop efficient techniques for constructing ORAMs that takes advantage of the SGX enclave technology. We demonstrate implementations of multiple ORAM schemes (linear, square root, and path ORAM) using Intel's SGX. We discuss the limitations of SGX as they pertain to implementing ORAM, and discuss alterations to the standard algorithms to overcome these limitations. We then evaluate the performance of our techniques.

1 Introduction

In a data outsourcing model using the cloud, an organization stores its data with a cloud provider instead of hosting it locally. Most providers charge by the gigabyte for both storage and transfer of data. In order to guarantee the confidentiality and integrity of the data, the organization will also encrypt it with a key not known to the cloud provider. While this prevents a potentially malicious provider from learning the contents of the data, they are still able to learn the *access pattern* by observing which data is accessed by the organization and when. This information can be used to glean sensitive information [1]. For example, it has been shown that by observing accesses to an encrypted email repository, an adversary can infer as many as 80 percent of the search queries [2].

Oblivious RAM (ORAM) schemes provide a solution to this problem. ORAMs were originally described using a CPU and memory model [3], but the adaptation to outsourced storage is straight-forward. The idea is based on a simple principle with a complex theoretical underpinning: Instead of the organization only accessing the data it wants with a particular request, it actually accesses multiple entries from the data store, and over time (or periodically) reshuffles its contents. The ORAM provides an abstraction through which the organization can access arbitrary parts of their data without leaking any information about which portions of data they are actually interested in.

One limitation of applying ORAM to the cloud is that ORAM schemes tend to assume that a client has a large local storage for a cache and the ability to easily and frequently download large portions of the protected dataset. These two parameters are directly at odds with why many organizations choose to outsource in the first place. This would hinder the adoption of ORAM for practical data outsourcing.

A new feature in Intel microprocessors, Software Guard Extensions (SGX), may provide a solution for this problem. SGX is a set of new instructions and modifications to the memory access architecture of Intel CPUs. SGX allows an application to create a container, referred to as an enclave, which is a protected area in the application's address space. The data inside this enclave is both confidential and secure, even in the face of an attacker with full control of the operating system. Attempted accesses to the enclave memory area from software not resident in the enclave are prevented at the hardware level [4].

In this work, we take advantage of the confidentiality and the integrity that SGX provides in order to build and compare ORAM systems in which the client has no involvement other than sending the read/write requests to the server. Under this model, the organization outsources their data as well as a secure enclave that manages the computational and bandwidth intensive tasks associated with ORAMs.

Building an SGX ORAM requires addressing major challenges. One challenge is the limited space available in the enclave, which could be a problem not only for processing data, but also for storing some auxiliary data structures such as permutation maps which are required by some ORAM algorithms. This limitation may also inhibit the execution of an ORAM shuffling algorithm. Another challenge is the absence of memory access obliviousness in the enclave, which requires additional arrangements in order to hide the memory access patterns. Such challenges are not encoun-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. This is a pre-print of 10.1016/j.cose.2019.101711

tered in traditional client-server models.

In this paper, we construct three different ORAMs using SGX: linear, square root, and path. We explore the limitations of SGX that make this task difficult, and discuss our modifications to the traditional algorithms in order to handle the enclave limitations. We also compare the performance of our SGX ORAMs and analyze the parameters that affect this performance. We identify the circumstances which make one type preferable over others.

2 Background

In this section we will discuss background information related to Intel's Software Guard Extensions (SGX) and oblivious RAM.

2.1 SGX

Under a traditional security model with an x86 processor, the system software (such as the operating system or virtual machine monitor) has full access to all system memory. As a consequence of this, it is effectively impossible for a process to prevent the system software from gaining access to its secrets. In a cloud computing model, this means that the outsourcing organization must trust the cloud provider with the contents of any processes running on the cloud servers.

SGX is a set of new instructions and modifications to the memory access architecture of Intel CPUs. SGX was first rolled out in the 6th Generation Intel® Core™ processor platforms (Skylake). SGX provides protection via enclaves which are protected areas in memory. Intel provides a collection of APIs, sample source code, libraries and tools that enable software developers to write applications that can make use of SGX.

An application with an enclave is split into at least two components: the trusted enclave component, and the untrusted application component. Control flow changes between enclave and untrusted code using ECalls (from application to enclave) and OCalls (from enclave to application). Data stored within the enclave memory (which is called Enclave Page Cache (EPC)) can only be accessed by enclave code, but the enclave itself is permitted to access memory from the untrusted component. (This facilitates the sharing of data between the trusted and untrusted components.)

The current version of SGX available in hardware is SGX1 (SGX2 [5] has been designed, but as of this writing has not be released in silicon.) One of the major limitations of SGX1 is that there is a hard limit. This memory is reserved on boot-up and cannot be changed at runtime. This limits the amount of data that can be protected inside an enclave without resorting to costly paging schemes, which has implications for applications being built to make use of SGX.

One area of research regarding vulnerabilities of SGX specifically focuses on side-channels [6], [7], [8], [9], [10], [11], [12]. A side-channel, in this case, is an unintentional leak of information from inside to outside the enclave. Our approach to side-channels is discussed in more detail in Section 4.2.

2.2 Oblivious RAM Overview

The goal of oblivious RAM is to hide the access pattern of data stored in a memory when the attacker has full access to the memory bus. It is motivated by the idea that an attacker with the access pattern can still potentially learn confidential information, even if the data in memory is always stored and transmitted in an encrypted form. The basic idea is that the memory access pattern can be hidden if we hide the addresses (indices) of the items which are requested by the client as well as the number of times each item is requested. We will now provide a brief overview of the three types of oblivious RAM studied in this work.

2.2.1 Linear ORAM

A trivial implementation for an oblivious RAM is to scan the entire memory for each actual memory access. This scheme is called linear ORAM. In a client-server setting, when the client needs to read item i, it requests all the items from the server then writes all values back to the server. This guarantees that an adversary on the server side will not be able to know the requested index nor the type of access. Clearly, this is a naive technique since each request will cost the client an overhead of $\mathrm{O}(N)$ accesses, where N represents the number of memory items.

2.2.2 Square Root ORAM

Goldreich and Ostrovsky [3] presented two ORAM constructions with a hierarchical layered structure: the first, Square-Root ORAM, provides square root access complexity and constant space requirement; the second, Hierarchical ORAM, requires a logarithmic space consumption and has polylogarithmic access complexity. Square-root ORAM was revisited in [13] to improve its performance in a secure multi-party computation setting.

Consider an ORAM A of size N. In order to prepare the data, we allocate an additional $\sqrt{N}$ dummy items and then randomly shuffle the $N + \sqrt{N}$ items and store them on the server. We also allocate a cache (called the shelter) at the client with size $\sqrt{N}$. When the client requests a specific item, it first checks its local shelter. If the item is there, then the client requests the location of a dummy item from the server. If the item is not there, then the client requests the location of the actual item from the server and stores a copy in the local shelter.

Accordingly, an adversary watching the server's memory access will see, for every $\sqrt{N}$ requests from the client, $\sqrt{N}$ accesses to random locations with at most one access to each of these locations. None of the actual locations that are needed by the client is revealed to the server since the server only sees the permuted locations. In addition, each location is guaranteed to be visited at most once.

After $\sqrt{N}$ accesses, a re-initialization process is required. Array A should be reshuffled again (obliviously) and the shelter items should be cleared.

2.2.3 PATH ORAM

Several recent ORAM schemes have been proposed [14], [15], [16], [17], [18], [19], [20]. The work of Shi et al. [21] adopted a sequence of binary trees as the underlying structure. Stefanov et al. [1] utilized this concept to build a simple ORAM called path ORAM.

In path ORAM we map every block of data in the ORAM represented by input array A to a (uniformly) random leaf in a tree (typically binary) on the server, using a position map which is stored at the client. Each node in the tree has exactly Z blocks which are initially dummy blocks. Each data item is stored in a node on the path extending from the leaf to which the data item was mapped, up to the root. When a specific item is requested, a position map is used to point out the leaf to which the block is mapped to. Then the whole path is read starting from the block's mapped leaf up to the root into a client side cache (the stash), from which the client can access the item it requested. After this, we remap the selected item to a different leaf node, and then the path just read is written back to the server.

3 Related Work

In this section we will discuss research efforts related to the usage of SGX for secure computation that are important in relation to this work. SGX has been used to implement a variety of secure computation techniques such as the private membership test [22], secure multiparty computation [23], secure indexing [24], and oblivious queries.

Most related to our work are other works that implement oblivious data stores using SGX. In a sense, these approaches can be thought of as use-cases for oblivious RAM systems. They differ in their interest of channel side attacks which may reveal some information about the enclave's memory access if no action is taken. Oblix [25] is an oblivious search index for encrypted data which hides memory accesses, hides the result size of searches, and supports insertions and deletions with good efficiency. Oblix also provides protection against modification attacks in which an attacker can modify data or queries. However, Oblix considers any side-channel leakage from within the enclave as out of scope. Obliviate [26] is a data oblivious filesystem which adapts the ORAM protocol to read and write data from a file within an SGX enclave. Obliviate utilizes Path ORAM protocol to achieve its target and even parallelizes its write-back process in order to boost the performance. While data are written to and read from files in [26], our work focuses on memory reads and writes. The work of [27] presents an SGX-based efficient SQL search over encrypted databases. The access pattern is protected against side channel attacks. The solution can process large databases without requiring any long-term storage on SGX. While requests are SQL statements in [27], they are memory requests in this work. OBFSCURO [28] utilizes SGX to transform a program into a side-channel-secure and ORAM-compatible one. Then, OBFSCURO ensures that its ORAM controller always performs data oblivious accesses. On the contrary to several works including this one which focus on data obliviousness, OBFSCURO presents a generic security framework against all memory-based side-channel attacks (code and data).

Most similar to our work is ZeroTrace [29](ZT), which implements and tests SGX implementations of path and circuit ORAM. There are a number of differences with our work. First, circuit and path ORAM are very similar in design and functionality. The primary difference lies in their eviction strategy. So, while ZeroTrace compares two very similar ORAM systems, our work compares the performance of three very different ORAM designs and studies the situations which favor each type. Another difference is the manner in which both works handle oblivious access of data inside the enclave. ZeroTrace makes use of the x86 `CMOV` instruction, a single instruction that only performs a move if a certain condition is met. Our approach is more general and applicable to a variety of systems. Other differences are related to the end-goal of each work. Our work focuses on studying and discussing a variety of issues related to implementing ORAM in SGX, such as oblivious shuffling, recursive position maps, and analyzing the specific design factors that affect performance such as number of auxiliary trees. ZeroTrace is more focused on system issues such as addressing integrity (active adversary) using a Merkle tree, supporting multiple back-end memory organizations (DRAM and HDD), and handling the interruption in the system by writing out the enclave state to untrusted storage.

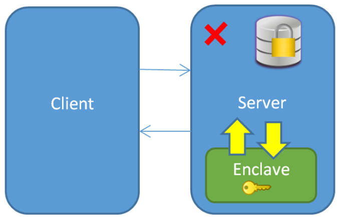

Fig. 1: The communication between the two parties. The red X indicates untrusted party. Encrypted Data are stored on the server. Encryption key is stored in the enclave.

4 Preliminaries

In this section we discuss our execution model, threat model, assumptions, terminology, and the limitations of SGX that create challenges for this work.

4.1 Execution Model

Our model has two parties: the client and the server. The server hosts both the enclave and the memory containing the ORAM data. The goal is to host an ORAM with no involvement from the client other than sending read/write requests. The purpose behind this goal is that we assume the client has limited processing power and limited memory resources. The server creates an SGX enclave, which is then considered part of the trusted computing base of the client. We assume that our data is an array of N items. Fig. 1 illustrates the presented model.

The client sends an encrypted read/write request (for a specific item) to the enclave host. The enclave host passes the message to the enclave. The enclave decrypts the message and executes the ORAM procedure on the server's memory, then returns the requested data to the client. In this model the client does not perform any of the ORAM processing, instead that work is done in the enclave. It is important to note that while the enclave hosts code and some cached ORAM data, the actual ORAM data is stored on the server outside the enclave. Data stored within the enclave is automatically encrypted using the memory encryption engine (MME) [30]. The main ORAM data, however, is stored encrypted in the server's main memory using a key derived by the software running inside the enclave. The MME is not involved in this part of the encryption.

4.2 Threat Model, Assumptions, and Constraints

We assume that our adversary has full privileges over the server (i.e., root privileges). We assume that the adversary will not perform a hardware attack against the CPU itself, but we do allow them to observe the memory bus at any time.

We adopt the standard security definition for ORAMs described in [31]. Accordingly, the target is to enable the client to read and write data blocks to and from the server without allowing an adversary to infer:

- Which items (blocks) have been requested by the client?
- What are the stored/read values?
- How many times has an item been requested?
- What type of request did the client issue (read or write)?

A secure channel can be constructed between the client and enclave. SGX provides a mechanism for the client and enclave to establish a secret key for a secure channel using remote attestation to share secrets. This remote attestation protocol uses a modified Sigma protocol to facilitate a Diffie-Hellman Key Exchange (DHKE) in order to establish the shared key [32]. We rely on the correctness of this protocol functions.

Memory accesses performed from within the enclave are page-level oblivious. Unfortunately, even though data stored within the enclave is encrypted before being stored in physical memory, oblivious access is not provided. This problem is distinct from the overall problem of using an enclave to provide an ORAM. Due to the fact that access to the enclave's memory is not oblivious, a scenario may emerge in which a malicious server could watch its memory bus to determine the memory addresses being accessed within the enclave. However, due to the design of the memory system on Intel CPUs, recent work [22] has found it reasonable to assume page-level obliviousness. Page-level oblivious means that accesses to two different memory addresses on the same physical memory page can be assumed to be indistinguishable. However, in order to protect against cache attacks that may be able to distinguish at a finer granularity than page-level, in our experimental section we also show performance results assuming cache line obliviousness instead of page-level obliviousness.

Known side channels *can* be prevented, but exhaustively analyzing them is outside the scope of this work. Recently, research into side-channel attacks against modern CPUs (as well the applicability of those side-channels to SGX) has been skyrocketing. The attacks target a variety of shared hardware resources on the CPU such as the cache [7], [8], [10], [12], page faults [11], branch target buffer [9], directional predictor [6], etc. Where straight-forward, our design incorporates protections against known side-channels. For example, we assume page-level obliviousness due to the existence of page fault side channels. (Although in the event that a side channel allows for a finer than page-level distinguishability of memory accesses, our results from Section 8.4 show how the performance of the system will be impacted.) We also ensure that procedures requiring oblivious execution can fit on a single page and require the same number of instructions to execute regardless of code path. (In algorithm 2 for example, the time consumption of every sequence is always the same.) In some cases, such as detailed branch side-channels [9], [6], mitigating their impact is not so straight-forward. While software based mitigations have been proposed by the authors, they come at a high performance cost. Given this, we assume that these mitigations could be applied to our techniques, but we consider a full analysis of their performance impact to be outside the scope of this work.

We consider this assumption regarding side channels reasonable primarily because side-channel mitigation is its own research area that is orthogonal to this work, and the prevention approaches being developed in this area can be applied to this work.

4.3 Terminology

A is an array of N items which should be stored on the server. Each item in A has a size of b bytes. E is the size of the enclave. The initialization process is the process of storing the initial values of A on the server. The identifier for an element (item) in A is its original index in A. Accordingly, the client requests an item by sending its index to the server. We denote a request from the client for a specific item in A using its index from the server as a read request and a request from the client to store an item in a specific index in A as a write request.

4.4 Challenges Due to Limitations of SGX

There are two major limitations of SGX that make this work challenging. In this section we will discuss those limitations. The details of how we overcome these limitations for each of our three ORAM systems are described in later sections.

4.4.1 Limited Memory Space

While it depends on the firmware, most SGX machines can allocate a maximum of 64 MB or 128 MB of memory to the enclave page cache (EPC) that is shared *among all enclaves*. This memory is allocated at boot time and cannot be changed. This means that, in the best case, our enclave has access to 128 MB of secured memory assuming a generous BIOS setting and no other enclaves running on the system. As such, our ORAM algorithms need to be modified to take into account this specific amount of available memory.

4.4.2 Lack of Memory Access Obliviousness

As mentioned in Section 4.2, memory accesses from within the enclave are only page-level oblivious. When the enclave is accessing its own memory, the page addresses being accessed can be seen by an attacker. As such, our ORAM algorithms will need to be modified to consider the obliviousness of their own internal memory accesses.

5 Building Linear ORAM using SGX

In this section, we show how to build a linear ORAM using SGX. Then we explain in details the implementation of square-root ORAM and path ORAM using our model. Linear ORAM is considered to be the trivial method for realizing ORAM. We demonstrate its SGX implementation here because it helps illustrate many of the challenges associated with constructing an ORAM using SGX.

5.1 Initialization

The initialization process is done as follows: Each item is encrypted using a block cipher (AES). The same key is used for all items, but the IV for each item is unique in order to ensure semantic security. IVs are chosen using a simple, incrementing counter. Throughout this process, the encrypted data is stored in memory on the server, but outside of the enclave.

5.2 Item Access

Under the simplest assumption, a lookup in a linear ORAM requires the client to read the entirety of the encrypted data and find the element it wants. By reading all of the data, obliviousness is maintained. Given that the entire contents of the ORAM cannot fit inside the enclave, the data is read in fixed size chunks less than the available memory size. Regardless of which chunk contains the requested items, all chunks are requested.

A client performs a memory access as follows:

1) The client sends a request to the enclave using the secure channel.
2) The enclave reads all data in chunks from the server's main memory (regardless of whether the client's request is a read or write).
3) For each chunk:
    a) The chunk is decrypted.
    b) For a read, the requested item is retrieved if it is available in the chunk.
    c) For a write, the given item is written if it should be stored in the chunk.
    d) The items in the chunk are re-encrypted using new IVs.
    e) The chunk is written back to the server's main memory.
4) The enclave returns the request item to the client, or in the case of a write, returns a dummy value.

It should be noted that even though this is an extremely inefficient ORAM implementation, using SGX means that almost the entirety of the data traffic and computation occurs at the provider's site (between the enclave and the data store), not at the client's. This is not the case in a standard ORAM implementation.

5.3 Complexity Analysis

The algorithmic complexity for data initialization is O(N). The complexity of a single memory access is also O(N).

5.4 SGX Influenced Design Choices

The limitations of SGX described in Section 4.4 influence our design in the following ways.

5.4.1 Limited Enclave Memory Space

The data is read into the enclave in chunks small enough to fit inside the enclave's limited memory space.

5.4.2 Lack of Enclave Memory Obliviousness

The lack of enclave obliviousness could be a major issue for even this simple ORAM. The core of the problem occurs in steps 3b and 3c above. An attacker monitoring the memory bus would be able to identify which chunk was read or written by simply noticing that the algorithm only performs a read or write during those steps if the current chunk contains the item being accessed. The attacker can also determine which page within the enclave contained the item, further narrowing down which item is it. This violates the basic protections provided by ORAM.

In order to overcome this issue, we have designed an approach to provide a basic, page-level oblivious memory access from within the enclave. Pseudo code for internally oblivious access is shown in Algorithm 1.

When performing a memory access (read or write) on a chunk, we ensure that exactly one item from every physical page is read (source pages), one item is written to the page containing the location of the result (destination page), and one item is written back to the source page again. Every page in a chunk has this exact access pattern regardless of whether the requested access is a read or a write or even if that page has the item requested. Consider lines 4 through 8 in Algorithm 1. If the source page contains the item we want, then we read that item and write it to `obv_bk[1]`, which is stored on the destination page. If the source page doesn't contain the item we want, we read any value and write it to dummy location `obv_bk[0]`, which is stored on the destination page. Since `obv_bk[0]` and `obv_bk[1]` are located on the same page, an attacker is unable to distinguish between writes to either one. This ensures the process of reading an item from the page and storing it in the destination memory location `obv_bk[1]` is oblivious. Next, in lines 9 through 15 we handle the situation where the access is a write. Here there are three scenarios: This is a write to this page, and we need to store the new block in the array; this is a read to this page, and we need to perform a dummy write by just writing back the value we read; this is an access to a different page and we need to write back the dummy value we read previously.

In summary, the algorithm guarantees that an adversary will not know which item has been read/written from within the enclave, and will not know whether a read or write has occurred. It is also immune to a side-channel timing attack since it always consumes the same amount of time.

6 SGX Square Root ORAM

When building a square root ORAM (described in Section 2.2.2) on SGX there are a number of challenges that need to be solved.

Algorithm 1 Internally Oblivious Memory Access From Within an Enclave

```
1: procedure $obliv\_access\_item(Arr, idx, size, new\_block, op)$
2:     $obv\_bk[2] = new\_block$
3:     for $z$ = 0 To $size$-1 do
4:         if $idx \geq z$ and $idx < z + PageSize$ then
5:             $obv\_bk[1] = Arr[idx]$
6:         else
7:             $obv\_bk[0] = Arr[z]$
8:         end if
9:         if $op \equiv' W'$ and $idx \geq z$ and $idx < z + PageSize$ then
10:            $Arr[idx] = obv\_bk[2]$
11:        else if $idx \geq z$ and $idx < z + PageSize$ then
12:            $Arr[idx] = obv\_bk[1]$
13:        else
14:            $Arr[z] = obv\_bk[0]$
15:        end if
16:        $z$ = $z + PageSize$
17:    end for
18:    return $obv\_bk[1]$
19: end procedure
```

1) Unless the number of items stored in the ORAM (N) is relatively small, then the data in the ORAM (A) must be passed to enclave in chunks since enclave memory is limited.

2) Shuffling requires storing a permutation map, and that map may be too large to fit inside the enclave.

3) Even if the permutation map were to fit inside the enclave, since we pass A in chunks, the permuted location may not be available inside the enclave at the appropriate time. For example, during a reshuffle, let us assume that only 30 items out of 120 items can fit inside the enclave and the item in location 15 is permuted to location 111. Since location 111 is currently not available in the enclave, we cannot move the item to its new location.

6.1 Initialization

At a high level, the initialization of a SQRT ORAM is fairly straightforward: The items in the ORAM are randomly shuffled, encrypted, and uploaded to the server. A permutation map, showing the location of each item, is then stored in the enclave and used as part of the lookup process later. With SGX, storing a permutation map inside the enclave is infeasible for large values of N, so an alternative is needed. As such, the items in the ORAM are randomly shuffled using an efficiently computable pseudo-random permutation (PRP) function [33]. By making use of a PRP, there is no need to store a full mapping containing the permuted locations of all items in the ORAM; they can be derived using the function. The function uses a randomly generated key which is stored inside the enclave.

We note that by using a PRP instead of a permutation map, we address the problem of accessing the permutation map obliviously. Traditional shuffling techniques such as fisher-Yates [34], the riffle shuffle [35] and Thorp shuffle [36] are not oblivious by design since an adversary can see the position that an element is shuffled to. An additional issue is that a traditional shuffling algorithm requires a permutation map, and accessing it to get the permutation value would leak the access location or require the map itself to be stored in yet another oblivious RAM. A PRP does not have this problem since the permuted value is a calculated value. However, even a PRP still leaves open the possibility of a side-channel attack which analyzes the code and data access patterns of the PRP itself. To eliminate this possibility, our PRP performs, for any value, the same number of steps using an array of two blocks, storing the "real" value in one block and all dummies in another.

6.2 Item Access

The general workflow of a data access in the SGX square root ORAM is:

1) The client sends the request to the enclave using the secure channel.
2) The enclave searches for the requested index in the shelter (which resides in the enclave not on the client as in traditional SQRT ORAM) using a technique that will be presented in Section 6.5. If it is not found, the enclave calculates the permutation value v_1 of the requested index and requests the item stored in v_1 from the server. If it is found, the enclave calculates the permutation value v_2 of a dummy index and requests the item stored in v_2 from the server. Since we know that a new initialization is needed after $\sqrt{N}$ accesses, it is guaranteed that every dummy item is accessed at most once.
3) The server reads the requested item and sends it to the enclave.
4) The enclave decrypts it. If the item has not been requested before, then it is stored in the shelter. In case of a write request, the value which was provided by the client is stored into the shelter. If the item has been requested before, then whatever

sent by the server is a dummy and we retrieve the requested item from the shelter.
5) The enclave sends the requested item to the client, or in the case of a write, returns a dummy value.

6.3 Re-Initialization

As the system executes, the enclave counts the accesses and reshuffles the ORAM after $\sqrt{N}$ accesses have occurred. This re-initialization process includes the following steps:

1) A new random key is created and stored in the enclave in order to calculate a new PRP.
2) The enclave reshuffles the contents of the ORAM using each item's new location according to the PRP.

Step 2 in this process is quite challenging with SGX because we assume the entire contents of the ORAM cannot fit into the enclave.

Given this restriction, special care must be taken during reshuffling. Several oblivious shuffling algorithms are available such as the Melbourne shuffle [37], AKS[38], Zig-Zag shuffle [39], randomized shellsort[40] and Batcher's network[41]. We chose the Melbourne shuffle since it has an access overhead of O($\sqrt{N}$), which is the best among all other algorithms. In addition, since the enclave is on the same machine as the server, the high message cost of $O(\sqrt{N})$ is not a big concern.

We make use of the Melbourne Shuffle with minor modifications for SGX. The details of our implementation can be found in Appendix A. The approach operates in two stages. First, the existing ORAM array A is processed as chunks (buckets of size $\sqrt{N}$ each) one chunk at a time into a larger, server-side temporary array T. Then, T is processed to become the newly shuffled ORAM, O. The process of creating T differs from a standard ORAM implementation because some elements (the visited elements) have their real values in the shelter, not on the server. Algorithm 2 shows our approach to processing the array A into T one chunk at a time while considering the values in the shelter. Oblivious writes are needed in lines 7 and 10 to prevent an adversary from knowing if the item is in chunk ch and if it is in the shelter. Accordingly, this hides if the item was requested by the client. Since all blocks are encrypted, the chunk is decrypted and processing T blocks start (lines 13 and 14).

Note that repeatedly requesting the same element i will reveal no information since for each $\sqrt{N}$ accesses, the server will access the permuted position of i only once: to fulfill the first request. After that, $\sqrt{N}$-1 distince dummy locations will be accessed before re-initialization since i is already in the shelter.

6.4 Complexity Analysis

The main time and space requirements of the enclave are the initialization/re-initialization requirements. If we ignore the cost of re-initialization for the moment, fulfilling client's requests requires O($\sqrt{N}$) time since the shelter should be scanned.

The details of our shuffling algorithm are available in Appendix A. An overview is available here. Our shuffle reads the input array A as buckets of size $\sqrt{N}$ into the enclave. For each bucket B, the algorithm prepares output

Algorithm 2 Filling T one Chunk at a Time

```
1: procedure $Reinitialize$(chunk $ch$)
2:     $obv\_bk[0]$ = first block in $ch$
3:     for every element $i$ in the shelter do
4:         $obv\_bk[1] = i$
5:         if $perm(obv\_bk[1].pos)$ belongs to $ch$ then
6:             encrypt $obv\_bk[1]$
7:             obliv-write $obv\_bk[1]$ back to $ch$
8:         else
9:             encrypt $obv\_bk[0]$
10:            obliv-write $obv\_bk[0]$ back to $ch$
11:        end if
12:    end for
13:    decrypt all items in ch
14:    fillT($ch$)
15: end procedure
```

blocks of size $p \log N$ each, where p is a constant. For the shelter, an array of size $\sqrt{N}$ items and an integer array of size $\sqrt{N}$ for storing items' indices are needed. Unfortunately, we need both $\sqrt{N}$ items arrays at the same time during the re-initialization process. Three extra short integer arrays are needed for Melbourne shuffle. One $p \log N$ output block is needed. Accordingly, the space requirements is O($\sqrt{N}$) items. We can estimate the number of items which the enclave can handle by solving the equation:

$$E = \sqrt{N} \times (2 \times b + 10) + \epsilon \quad (1)$$

where E is the size of the enclave, b is the size of an item, and ϵ is the size of some extra variables which are needed in our routines such as obv_bk (Algorithm 2). Assuming that the enclave can handle only 64MB, the size of an item is 16 bytes, and ϵ is 1000 bytes, the enclave can handle cases with $\sqrt{N}$ = 1.5 million items. So, unless the server has a storage limitation, the enclave can handle cases with N= 10^{12} items.

Let ℓ be the time cost of one oblivious access inside the enclave. Since $\sqrt{N}$ is the maximum size of an array in the enclave and $\ell \leq \sqrt{N}/PageSize$, the access overhead for our technique is O($N/PageSize$).

6.5 SGX Influenced Design Choices

We will now discuss the SGX specific challenges for our square root ORAM.

6.5.1 Limited Enclave Memory Space

There are two places in this algorithm where we made choices in order to accommodate the memory constraints of the enclave.

- *Shelter.* We made sure that in all stages, the amount of memory that is needed in the enclave is around $\sqrt{N} \times (2 \times b + 10)$.
- *Permutation Map.* The limited memory space made it infeasible to store a permutation map inside the enclave. Instead, we made use of a pseudo-random-permutation function.

6.5.2 Lack of Enclave Obliviousness

There are two instances where internally oblivious access is required.

- *Shuffling.* As already mentioned, existing shuffling algorithms are not designed to be oblivious. Our oblivious Melbourne shuffle, however, ensures that all relevant reads and writes to $\sqrt{N}$ chunks, $counter\_Accs$, $counters$, and $ptrs$ are oblivious. To prevent an adversary from concluding the number of items in stage 1, we made each iteration require $p \log N$ steps regardless of the number of "real" items in each block.
- *Item Access.* During item access, a search of the shelter must occur inside of the enclave. This search needs to be done obliviously so as not to reveal the identity of the shelter item found. Algorithm 1 (explained in linear ORAM) is not suited for this, as it assumes that the items being searched are stored sorted by index. In this case, the shelter items can be stored in any order. As such, a modified algorithm is available in Algorithm 3 which searches each entry for the item. The same design principals that make Algorithm 1 page-level oblivious also apply to this algorithm. In Algorithm 3, $obv\_bk[0]$ is used for dummy accesses and $obv\_bk[1]$ for the real item. If the operation is a write, then element z is replaced with the new value, otherwise, z is replaced by one of obv_bk two values (lines 10-16).

Algorithm 3 Internally Oblivious Memory Search From Within an Enclave

```
1: procedure $obliv\_access\_item(Arr, idx, size, new\_block, op)$
2:     $obv\_bk[2] = new\_block$
3:     $obv\_bk[1] = NOT\_FOUND$
4:     for each element $z$ in $Arr$ do
5:         if $idx = z$ then
6:             $obv\_bk[1] = idx$
7:         else
8:             $obv\_bk[0] = z$
9:         end if
10:        if $idx = z$ then
11:            $z = obv\_bk[1]$
12:        else
13:            $z = obv\_bk[0]$
14:        end if
15:        $z = (op \equiv' W'$ and $idx = z) \times obv\_bk[2] + (op \equiv' R'$ or $idx \neq z) \times z$
16:    end for
17:    return $obv\_bk[1]$
18: end procedure
```

6.6 Security Analysis

In this section we present a step-by-step security analysis for a request. We assume that the key obtained by the attestation process is secure and that the encryption/decryption algorithm implementation (AES) for a request is side channel resistant.

We show that the initialization does not leak information:

- In the first stage of initialization, Algorithm 2 guarantees that returning visited items back to A's chunk (ch) consumes the same amount of time no matter how many items are in the shelter since the shelter is always scanned linearly. ch is, at this stage, securely populated with real items which were in the shelter.
- The number of "real" items to be filled in a T block is not leaked since dummy operations are added in Algorithm 6 and all data structures in it does not leak information since they are accessed obliviously $\sqrt{N}$ times.
- Stage 2 also leaks no information since each T block will be linearly scanned and all accesses to the output array are oblivious.

When there is a new request, Algorithm 3 scans the shelter in a fixed time since the shelter has a fixed size and the number of executed statement is always the same. Line 15 hides the type of operation.

The result of shelter scanning would lead to the decision of sending either a dummy request or a "real" request to the server. Such a choice is hidden by using using the two elements of obv_bk, which are contained on the same physical page.

When the enclave receives the item from the server, it has to be stored in the shelter which is done using one oblivious write.

Accordingly, for two sequence of requests $seq1$, $seq2$ with the same number of requests, an adversary has no way of distinguishing between them despite the fact that they have different memory access pattern. An adversary has no way of linking one input with a specific memory access patterns. Despite the fact that our implementation has if-else statements, all such statements are balanced in order to eliminate side channel attack possibilities [42].

7 SGX Path ORAM

When building a path ORAM on SGX there are a number of challenges to be solved:

- Since the memory access of the enclave is not oblivious, an adversary may be able to guess which node in the tree had the item by watching the memory access of the stash.
- Ideally, Path ORAM assumes that a position map can fit in the client memory. Since it requires O(N) space, this may be a problem with SGX. An alternative approach to storing the position map is required.

7.1 Initialization

After creating items' tree and other auxiliary trees (if needed) by the server, the enclave creates a position map with the size of a threshold Pos_Map_Limit which we assume is known to all parties. The enclave encrypts all trees. After creating the position map, the enclave fills the position map (recursively if necessary) and then initializes items one by one.

Pseudo code is shown in Algorithm 4. Every tree has a stash with the same id. Function `add_to_stash(`$tree\_id$`, `i`, `$value$`, `new_path`)` adds item i to the stash of tree $tree\_id$ and store the id of its new path with it (new_path). We define the id of a path by the number of the leaf with which such path starts. $value$ is also stored in i carrying the return value from the recursive function (i.e. the new path for i in the next tree). In other words, the item is stored in the stash in addition to its new path id in the current tree and its

Algorithm 4 Initialization for path ORAM

```
procedure $initailization$
    $l$ = Number of trees
    for $i$ = 1 to size_of_positionmap do
        if there are auxiliary trees then
            positionmap[$i$] = $init\_rec$($i$,$tr_l$,random_value)
        else
            positionmap[$i$] = random_value
        end if
    end for
    for every item $i$ in Array $A$ do
        $path$ = random_value
        $new\_path$ = random_value
        add_to_stash(0,$i$,$val\_i$,$new\_path$)
        set_position_map($i$,$new\_path$);
        read_path($tr_0$, $path$)
        write_path($tr_0$,$path$)
    end for
end procedure

procedure $init\_rec$($i$,$tr_l$,$path$)
    $i1$= $i$×2
    $i2$= $i$×2+1
    $new\_path$ = random_value
    read_path($tr_l$, $path$)
    if $tr_l$ is not $tr_1$ then
        $v1$ = init_rec($i1$,$tr_{l-1}$,$random\_value$)
        $v2$ = init_rec($i2$,$tr_{l-1}$,$random\_value$)
    else
        $v1$ = random_value
        $v2$ = random_value
    end if
    add_to_stash($l, i1, v1, new\_path$)
    add_to_stash($l, i2, v2, new\_path$)
    write_path($tr_l$,$path$)
    return $new\_path$
end procedure
```

new path id in the next tree (tr_{l-1}) which is returning from the recursion. Note that the read and write path methods are applied to the old path of the item, not the new one.

Function $init\_rec$ initializes every element in the position map (lines 4-8). The recursion continues in $init\_rec$ until reaching tr_1 which includes the position map of the items' tree (tr_0).

The path id for a specific item in tree tr_l is needed in tree tr_{l-1}, but we also store it in the stash for tree tr_l. This is to eliminate the need for expensive calls to $get\_position\_map$ in every write path call. Functions $set\_position\_map$ and $get\_position\_map$ have almost the same structure. They both call the recursive function get_pos_rec which will reassign targeted items to new paths. Pseudo code for $get\_position\_map$ and $set\_position\_map$ is shown in algorithm 5. $v1$ in line 38 is the future path id of $i1$ in the next tree tr_{l-1}, while $pos1$ is the current path id of $i1$ in the next tree (tr_{l-1}). While new_path is the new path id of both $i1$ and $i2$ in the current tree. new_pos is a public variable which carries the final result for $get\_position\_map$.

7.2 Item Access

Assuming all trees have been initialized, a request can be fulfilled as follows:

1) The client sends the request to the enclave using the secure channel.
2) The enclave uses $get\_position\_map$ (Algorithm 5) to determine the path (directly or recursively) that has the requested item. Line 11 in $get\_position\_map$ function should be executed obliviously, otherwise, an adversary can detect the requested index.
3) Read path procedure: reads all nodes on the path returned from $get\_position\_map$ into the stash.
4) The requested item is assigned to a new path. Line 25 in $set\_position\_map$ (Algorithm 5) should be executed obliviously, otherwise, an adversary can detect the requested index.
5) The requested item is sent to the client.
6) Write path procedure: write items into nodes starting from the leaf to the root. For every item i in the stash, we check if i is mapped to a path which intersects with the current node v. This checking is done by knowing the current level in tree and the initial leaf which the path starts from. Accordingly, you can easily determine the range of leaves that have v in their paths.

If i is mapped to a path which intersects with the current node v, we move it to a temporary node tmp which is initially filled with dummies. If all Z dummies in tmp are replaced with real items, tmp is encrypted and copied to v. Then we move upward to check the next node in the path.

7.3 Complexity Analysis

As mentioned before, the access overhead for the traditional version of this technique is $(\log N)^2$. The time cost of one oblivious access to the position map is $Pos\_Map\_Limit/PageSize$. Accordingly, the access overhead is O($(\log N)^2$ $Pos\_Map\_Limit/PageSize$).

7.4 SGX Influenced Design Choices

The limitations of SGX described in Section 4.4 influence our design in the following ways.

7.4.1 Limited Enclave Memory Space

Due to the enclave's memory limitations, in many cases the entire position map cannot be stored inside the enclave. As such, we adopted a recursive position map described in [31]. This involves storing the position map itself in another ORAM, $O1$, and storing the position map for $O1$ at the client. If this position map is still too large, then another ORAM $O2$ is built to store the position map for $O1$, etc. Fig. 2 shows an example for mapping i=4.

We will call these additional trees auxiliary trees. This recursive approach reduces the efficiency in terms of both space and time, as it requires additional queries to the server and a larger number of auxiliary trees stored on the server. This trade-off is required in order to overcome the SGX limitation. It may be tempting, instead of using the recursive technique, to store the (encrypted) position map on the server and to process it in a streaming fashion during a request. However, that would degrade the performance since processing each request would require O(N/E) reads, where E is the size of the enclave. Each batch will be then scanned linearly (page by page) inside the enclave.

Algorithm 5 *get_position_map* And set_position_map

```
1:  procedure get_position_map($i$)
2:      if there are no auxiliary trees then
3:          return positionmap[$i$] (Obliv)
4:      else
5:          $index = i$
6:          for every auxiliary tree do
7:              index = index / 2
8:          end for
9:          $l$ = Number of trees
10:         $original\_i = i$
11:         positionmap[$i$] = $get\_pos\_rec(tr_l$,index, positionmap[$i$],-1) (Obliv)
12:     end if
13:     return $new\_pos$
14: end procedure
15: procedure set_position_map($i$,$new\_value$)
16:     if there are no auxiliary trees then
17:         positionmap[$i$] = $new\_value$ (obliv)
18:     else
19:         $index = i$
20:         for every auxiliary tree do
21:             index = index / 2
22:         end for
23:         $l$ = Number of trees
24:         $original\_i = i$
25:         positionmap[$i$] = $get\_pos\_rec(tr_l$,index, positionmap[$i$],$new\_value$) (Obliv)
26:     end if
27: end procedure
28: procedure get_pos_rec($tr_l$,$i$,$path$,$new\_value$)
29:     $i1 = i \times 2$
30:     $i2 = i \times 2+1$
31:     $new\_path$ = random_value
32:     read_path($tr_l$, $path$)
33:     if $tr_l$ is not $tr_1$ then
34:         $temp$=obliv_access_item(stash $l$,$i1$,size of stash $l$,Null,'R')
35:         $pos1 = temp$.value
36:         $temp$=obliv_access_item(stash $l$,$i2$,size of stash $l$,Null,'R')
37:         $pos2 = temp$.value
38:         $v1 = get\_pos\_rec(tr_{l-1}, i1, pos1, new\_value)$
39:         $v2 = get\_pos\_rec(tr_{l-1}, i2, pos2, new\_value)$
40:         store $i1$ and $i2$ back in stash $l$
41:     else
42:         $temp$=obliv_access_item(stash $l$,$original\_i$,size of stash $l$,Null,'R')
43:         $new\_pos\_original\_i = temp$.value
44:         $temp$.value = $temp$.value $\times (op \equiv' R') + new\_value \times (op \equiv' W')$
45:     end if
46:     write_path($tr_l$,$path$)
47:     return $new\_path$
48: end procedure
```

7.4.2 Lack of Enclave Memory Obliviousness

There are two times during the path ORAM algorithm that we need to consider enclave oblivious memory accesses. First, when getting/setting the leaf id from the position map, we need to make use of Algorithm 1. Second, when searching for an element in the stash we need to make use of Algorithm 3.

7.4.3 Security Analysis

In this section we present a security analysis of our approach. We assume that the key obtained by the attestation process is secure and that the encryption/decryption algorithm implementation (AES) for a request is side channel resistant. These are the same assumptions made for the analysis of the SQRT ORAM.

- During initialization, the position map is filled with random data. In terms of security, initialization can be seen as issuing N requests. Accordingly proving that a request doesn't leak any information and is not distinguishable from another request, in term of memory access pattern, is satisfactory to prove that initialization is secure.
- When an enclave receives a request, its access to the position map to find the path ID is performed obliviously.
- When the server reads a path, the enclave stores it obliviously in the stash item by item using `add_to_stash` function mentioned in Algorithm 4. Since both the number of items in a node and the height of the tree are constants, no information may be leaked when a path is read.
- When a write-path is executed, exactly one scan of the stash is required for each node. Therefore, an adversary has no way to guess the number of "real" items in a node.
- In Algorithm 5, an adversary can differentiate between the processing of each tree since the if-else statement in line 33 is not balanced, however, such information has no benefits to an adversary. The processing of each tree per se is resistant to timing side channel attacks.

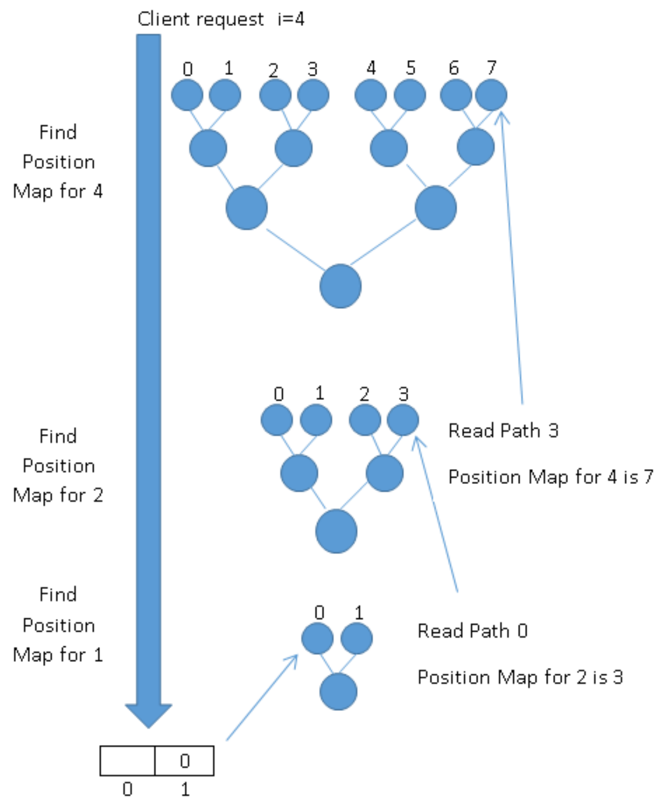

Fig. 2: Recursive Position Map

Overall, since accessing the position map, read-path, write-path and recursive technique don't reveal information, two sequences of requests with the same length are not distinguishable despite the fact that they may have different memory access (different tree paths).

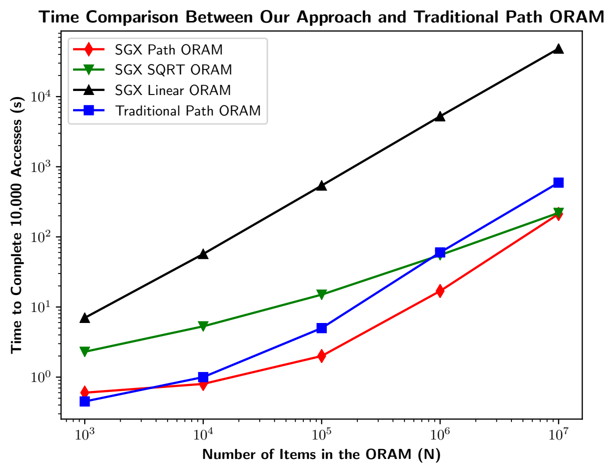

Fig. 3: Time Comparison Between SGX ORAMs and Traditional Path ORAM

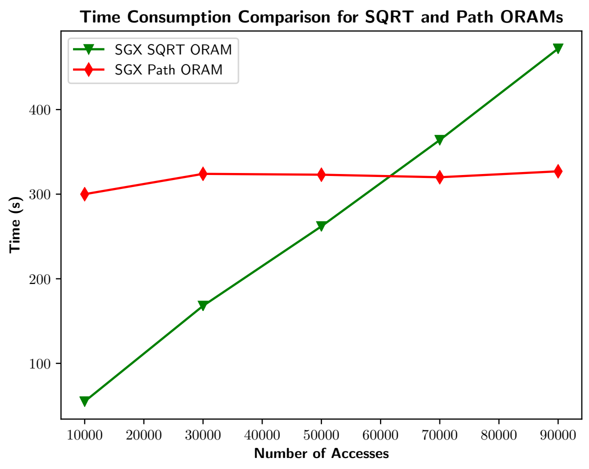

Fig. 4: Time comparison for SGX SQRT and SGX Path ORAMs. N (number of items) is 10^6 in all experiments.

8 Experimental Evaluation

In this section we will show the results of testing the implementation of our enclave-based ORAM techniques.

8.1 Experimental Setup

We implemented SGX-based linear, SQRT, and path ORAMs[1]. Our implementations are built on the top of Intel's SGX SDK version 1.9 as well as Intel's IPP cryptographic library. Our tests were run on a machine with 32 GB RAM and an Intel(R) Core(TM) i7-6770HQ 2.60GHz CPU with an enclave's maximum size of 128 MB. In all our tests, Z=4 (the number of blocks inside each node) unless it is mentioned otherwise. The size of a data block is 16b unless mentioned otherwise. In order to determine our stash size for Path ORAM, we look to existing work [1] showing that when $Z \geq 4$, the stash size is to be bound to $\omega(\log N)$ [1]. As such, we used stash size of 10000 items in general and 100000 when $N \geq 10^7$, which was enough not to overflow in any test. In SQRT ORAM, the number of reshuffling clearly depends on N. When $N = 10^6$ for example, the reshuffle is done every 1000 ($\sqrt{10^6}$), so 10000 accesses requires 10-11 shuffles. If $N = 10^7$ then reshuffling occurs every 3163 accesses which means 3-4 shuffles during the experiment.

During performance tests there are always two processes running: The client and the server of Fig. 1. The client process initiates access requests, while the server process initializes and passes traffic to and from the enclave.

Our results include the time for initialization of the ORAM as well as the time for data accesses.

8.2 Overall Performance

Our first set of tests show the overall performance of our implementations. We include 4 ORAM implementations: our three SGX ORAMs, and finally our own implementation of a non-SGX path ORAM in which all stashes and the position map are stored on the client. In this case, the client issues all read- and write-path commands to be executed on the server. Unfortunately, the source code for OBLIVIATE [26] is not available so we could not include it in the comparison, while [28] and [27] have different scopes of work.

Fig. 3 shows the elapsed time for 10,000 accesses using all four implementations. As expected, among the SGX ORAMs, linear ORAM has the worst performance for all of our testing. This is different from the findings of other traditional ORAMs in which linear ORAM has the best performance when N is smaller than a break-even point [3]. This is due to the fact that the communication cost between the client and the server is relatively high in traditional non-linear ORAMs while all ORAM management is done on the server in our model.

The traditional path ORAM has a curve similar to that of the SGX path ORAM, but is still outperformed by the implementation which includes the enclave despite the fact that each request in the SGX implementation incurs the AES encryption/decryption overheads in addition to the costs of moving memory across the enclave boundaries (ECALL/OCALL). This is due to the fact that all ORAM procedures are executed by the server instead of the client. This experiment should be considered the best case scenario for the traditional path ORAM, as the client and server are located on the same physical machine. The performance difference would be further exacerbated if the client and server were separated by a slow Internet link due to fact that in the traditional approach the client itself handles the steps for reads and writes.

8.3 SGX Path vs SGX SQRT ORAM

We also compared the performance of the SGX Path and SQRT ORAMs when the ORAM size (N) is fixed and the number of items being accessed varies. The reason this is interesting is that it can illustrate the performance trade-off between the two with regards to how each handles the initialization of the ORAM data. Fig. 4 shows the results. As can be seen, the performance of the SQRT ORAM depends heavily on the number of accesses, while the performance

1. The source code of our implementation will be made available coinciding with the publication of this work.

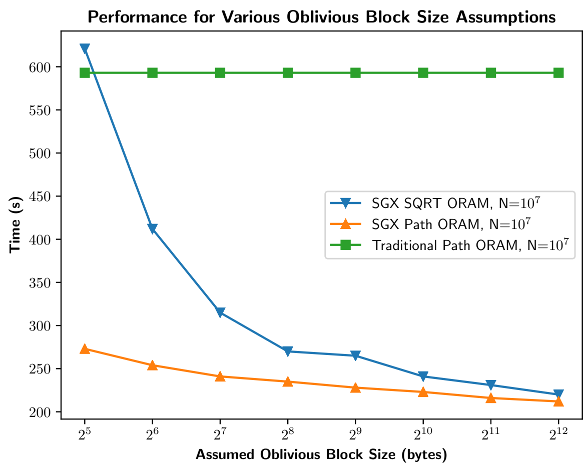

Fig. 5: Performance for Various Oblivious Block Size Assumptions

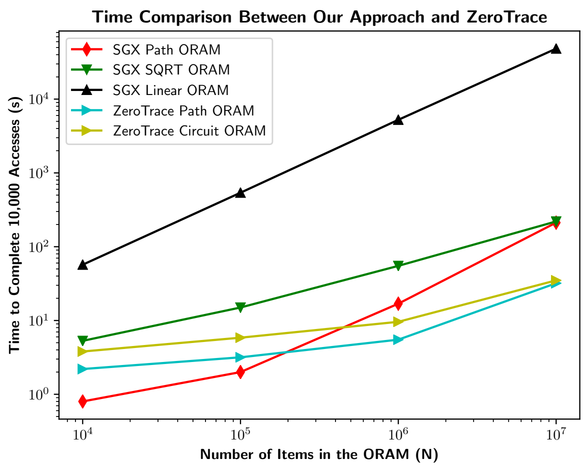

Fig. 6: Time Comparison Between Our Approach and ZeroTrace

of path ORAM does not. The reason is that path ORAM has a very intensive initialization process following by very fast individual accesses. The difference in magnitude between the initialization time and access time is so large that in this experiment the number of accesses had no discernable impact on the performance. SQRT ORAM, on the other hand, better amortizes the cost but it has a higher overall overhead. This means that for a small number of accesses SQRT ORAM is faster, but after a certain point path ORAM will lead to significant performance gains.

8.4 Effects of Page-Size Assumption

Although our work assumes that the SGX protected memory has page-level obliviousness, we recognize that recent work in side-channels indicates that a finer granularity may be required. Changing this assumption will impact the algorithm performance because the oblivious access portions of the algorithms will need to operate on smaller block sizes. For example, Algorithm 3 needs to run for each oblivious block of data, so smaller block sizes will result in the algorithm running more. We investigated the performance impact of assuming obliviousness for smaller block sizes.

Fig. 5 shows the performance of the SGX SQRT and SGX path ORAMs with $N = 10^7$, assuming different granularities for assumed obliviousness. As can be seen, the assumed obliviousness granularity impacts the SGX SQRT ORAM much more than the SGX path ORAM. For comparison purposes, Fig. 5 also includes the performance of the non-SGX, traditional path ORAM for this N value. Given that the traditional ORAM makes no attempts at obliviousness, assumed obliviousness has no impact.

8.5 Additional Path ORAM Tests

We study the effect of the size of position map (and consequently, the number of auxiliary trees) on the performance of path ORAM by using different thresholds for the size of position map. Table 1 shows expected negative correlation between time and the threshold of the position map. However, the results suggest that unless the increase in threshold decreases the number of trees, no significant improvement in performance is noticed.

8.6 Comparison to ZeroTrace

In this section, we compare our system to ZeroTrace. For our first experiment, we measured the time to both initializing the ORAM and performing 10,000 accesses on it for a variety of ORAM sizes. Fig. 6 shows the results. As can be seen, ZeroTrace outperforms our SGX path ORAM implementation for larger N values, while our approach is faster for smaller N values. The reason for this is that, ZeroTrace is generally faster when it comes to initializing the ORAM. Our faster results for $N \leq 10^5$ is due the fact that our system doesn't need to use a recursive position map for small N values, while ZeroTrace always uses a recursive position map. For larger N values both systems are using a recursive map, and ZeroTrace's faster initialization time bears fruit.

Next, we analyzed the time taken to perform the accesses after ORAM initialization has occurred. Table 2 shows these results. As can be seen, our approach has faster access times than ZeroTrace.

Given these two results, we performed an additional experiment where we held the number of items in the ORAM constant at $N = 10^6$ (just above the cross-over point in Fig. 6) and varied the number of accesses. We considered the total time to initialize the ORAM and perform the accesses. The results are shown in Fig. 7. As can be seen, the results are consistent with our previous findings. For a small number of accesses (10,000), the initialization time is a larger part of the overall time, and ZeroTrace is slightly faster than our approach. As the number of accesses increases, however, our advantage grows.

We summarize our contribution over ZeroTrace as follows:

- Despite the fact that ZeroTrace implements two ORAMs (path and circuit), they are almost the same. The main difference is the eviction policy. We have implemented three different ORAMs (path, SQRT, and linear) with different challenges.

N	Size of position map	Trees	Total Time	Initialization Time
1×10^6	1×10^5	3	445	441
1×10^6	2×10^5	2	383	381
1×10^6	5×10^5	1	300	298
1×10^6	9×10^5	1	327	324
2×10^6	2×10^5	3	1,631	1,627
2×10^6	5×10^5	2	1,440	1,436
2×10^6	9×10^5	1	1,289	1,284
3×10^6	2×10^5	4	6,624	6,617
3×10^6	5×10^5	3	5,403	5,399
3×10^6	9×10^5	2	3,202	3298

TABLE 1: Time consumption (in seconds) for SGX's path ORAM for 10,000 accesses with different position map thresholds.

N	ZT Path	ZT Circuit	Our Path
10^3	T	T	0.05
10^4	0.15	0.31	0.05
10^5	0.23	0.49	0.06
10^6	0.32	0.72	0.08
10^7	0.41	0.9	0.3

TABLE 2: Access Times for our path ORAM, ZeroTrace (ZT) path and circuit ORAMs. They are calculated by finding the average access time of 10000 accesses. Initialization time is excluded when calculating these averages. T indicates a termination in the program.

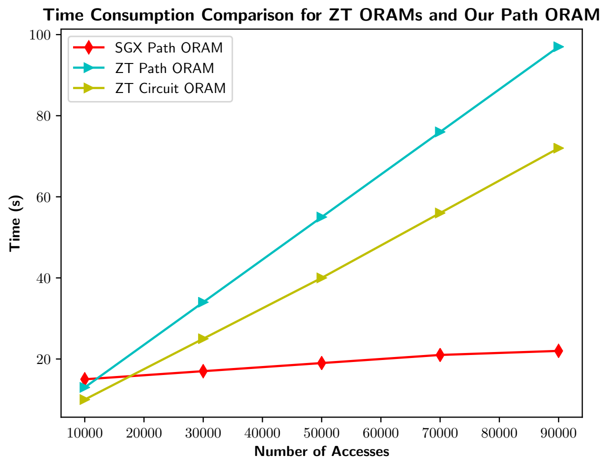

Fig. 7: Time Comparison Between ZeroTrace ORAMs and Our Path ORAM using Different Numbers of Accesses. N is 10^6.

- While we rely on a page-level obliviousness in many parts of our implementations, ZeroTrace relies on the `cmov` instruction which is limited to x86 assembly. We have tackled several different challenges in implementing obliviousness when utilizing page level obliviousness.
- As we show in the experimental section, our access time outperforms ZeroTrace when our position map is utilized (N is not big enough to activate the recursive technique), approximately 10^7 items.

9 Future Work

This work takes traditional ORAM, which has two parties (the client and server), and expands it to three parties (the client, the enclave, and the server). We have taken special care to ensure obliviousness between the enclave and the server, but future work should consider potential information leakage between the client and the enclave. There are a few important aspects of this to consider:

1) Size of the query and response. If the client's query and/or the server's response varies in size from query to query, then this could be problematic. The obvious solution (and the one we will claim in this work) is simply to ensure that all queries and replies are constant size. It would be interesting to further investigate this and determine if other approaches could be used that might allow more flexible queries and responses.
2) The timing of the query and response. There are a variety of potential timing side-channels to consider here. One, which has been handled in this work, is how long it takes the server to produce a response. Another which we have not considered is how much time passes between consecutive queries from the client (assuming that a client is processing the result of one query and using it to somehow produce another query). Can these timings reveal information about the operation and data the client is using? Is the problem severe enough that the entire client would also need to be executed obliviously, negating the positive impact of moving ORAM into SGX? These are interesting questions deserving of future investigation.

10 Conclusion

In this work we analyze three oblivious RAM algorithms, adapting them and implementing them using Intel's SGX. While SGX provides a way to protect the confidentiality of the contents of the enclave, its limited enclave memory size of lack of oblivious access to enclave memory required adaptations to the existing ORAM algorithms. By making use of SGX, we can shift almost the entirety of the computation and network traffic from the client to the enclave, significantly lowering the effective overhead of using an ORAM.

Our experimental results show that SQRT ORAM is a good option for datasets with a large number of items and infrequent accesses. Path ORAM, on the other hand, is a preferable option for smaller datasets and with larger numbers of access requests. Using today's SGX technology, SQRT ORAM can handle up to 16×10^{12} bytes of data with a 64 MB enclave's maximum size, while path ORAM practically has no limit for the size of data since the stash can be recursively stored.

Acknowledgement

This publication was made possible by the NPRP award NPRP X-063-1014 from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] 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 *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 299–310.

[2] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation." in *NDSS*, vol. 20, 2012, p. 12.

[3] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[4] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *HASP@ ISCA*, 2013, p. 10.

[5] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel®; software guard extensions (intel®; sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, ser. HASP 2016. New York, NY, USA: ACM, 2016, pp. 10:1–10:9. [Online]. Available: http://doi.acm.org/10.1145/2948618.2954331

[6] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 693–707. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173204

[7] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 490–505.

[8] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[9] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security*, 2017, pp. 16–18.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[11] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897885

[12] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 990–1003.

[13] 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 *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 218–234.

[14] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious ram practical," *MIT-CSAIL-TR-2011-018*, 2011.

[15] K.-M. Chung and R. Pass, "A simple oram," CORNELL UNIV ITHACA NY, Tech. Rep., 2013.

[16] I. Damgård, S. Meldgaard, and J. Nielsen, "Perfectly secure oblivious ram without random oracles," *Theory of Cryptography*, pp. 144–163, 2011.

[17] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in) security of hash-based oblivious ram and a new balancing scheme," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2012, pp. 143–156.

[18] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 139–148.

[19] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing oram and using it efficiently for secure computation," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2013, pp. 1–18.

[20] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 182–194.

[21] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o ((logn) 3) worst-case cost," in *International Conference on The Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 197–214.

[22] S. Tamrakar, J. Liu, A. Paverd, J.-E. Ekberg, B. Pinkas, and N. Asokan, "The circle game: Scalable private membership test using trusted hardware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 31–44.

[23] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, "Secure multiparty computation from sgx," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 477–497.

[24] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "Hardidx: practical and secure index with sgx," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2017, pp. 386–408.

[25] 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 (SP)*, May 2018, pp. 279–296.

[26] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A Data Oblivious File System for Intel SGX," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[27] S. Cui, S. Belguith, M. Zhang, M. R. Asghar, and G. Russello, "Preserving access pattern privacy in sgx-assisted encrypted search," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–9.

[28] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on intel sgx." NDSS, 2019.

[29] S. Sasy, S. Gorbunov, and C. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[30] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[31] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[32] Intel®, "Sgx remote attestation," https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, Jul. 2016.

[33] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2014.

[34] D. E. Knuth, "Seminumerical algorithm (arithmetic)," *The art of computer programming*, vol. 2, 1969.

[35] D. Aldous and P. Diaconis, "Shuffling cards and stopping times," *The American Mathematical Monthly*, vol. 93, no. 5, pp. 333–348, 1986.

[36] E. O. Thorp, "Nonrandom shuffling with applications to the game of faro," *Journal of the American Statistical Association*, vol. 68, no. 344, pp. 842–847, 1973.

[37] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The melbourne shuffle: Improving oblivious storage in the cloud," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2014, pp. 556–567.

[38] M. Ajtai, J. Komlós, and E. Szemerédi, "An 0 (n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. ACM, 1983, pp. 1–9.

[39] M. T. Goodrich, "Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o (n log n) time," in *Pro-*

Algorithm 6 Filling T as part of the Melbourne Shuffle

```
procedure FILLT(chunk $ch$)
    if $ch$ is the first chunk of data then
        create new secret (key for permutation)
    end if
    $sqrt\_of\_total = \sqrt{N + \sqrt{N}}$
    for every element $i$ in $ch$ do
        $z = perm(i.pos)/sqrt\_of\_total$
        obliv-increment $counters[z]$
        obliv-increment $counterAccs[z]$
    end for
    for $j$=2 to $sqrt\_of\_total$ do
        $counterAccs[j] + = counterAccs[j-1]$
    end for
    for every element $i$ in $ch$ do
        $z = perm(i.pos)/sqrt\_of\_total$
        obliv-read $counterAccs[z]$ to $index$
        obliv-write a pointer to $i$ in $ptrs[index]$
        obliv-decrement $counterAccs[z]$
    end for
    $curloc$ = 0
    $counter$=0
    $obv\_bk[0]$=dummy
    for $i$=1 to $sqrt\_of\_total$ do
        for $j$= 1 To $size\_of\_T\_block$ do
            if $counter < counters[i]$ then
                obliv-read $ch[ptrs[j+curloc]]$ to $obv\_bk[1]$
                $T[counter++]$ = $obv\_bk[1]$
            else
                obliv-read random block to $obv\_bk[1]$
                $T[counter++]$ = $obv\_bk[0]$
            end if
        end for
        $curloc+ = counters[i]$;
        $counter$=0;
        encrypt $T$ block
        store $T$ block on server
    end for
end procedure
```

ceedings of the 46th Annual ACM Symposium on Theory of Computing. ACM, 2014, pp. 684–693.

[40] ——, "Randomized shellsort: A simple oblivious sorting algorithm," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010, pp. 1262–1277.

[41] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[42] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 15–26.

APPENDIX A
SGX MELBOURNE SHUFFLE

While the details of the Melbourne Shuffle can be found in [37], we provide a description here of the SGX-safe version implemented for this work.

Let A be the input array and O be the output array. Both arrays have the size of N items. Let T be a transitional array in the server. T has the size of $Np \log N$ items where p is a constant. The Melbourne shuffle has two stages:

- *Stage 1: Filling T out of A.* The idea is to read the input array A as buckets of size $\sqrt{N}$ into the enclave. For each bucket B, we prepare output blocks of size $p \log N$ each. Each block includes all the items in B that are permuted to the corresponding bucket BT in array T where BT has the size of $p \log N \sqrt{N}$.

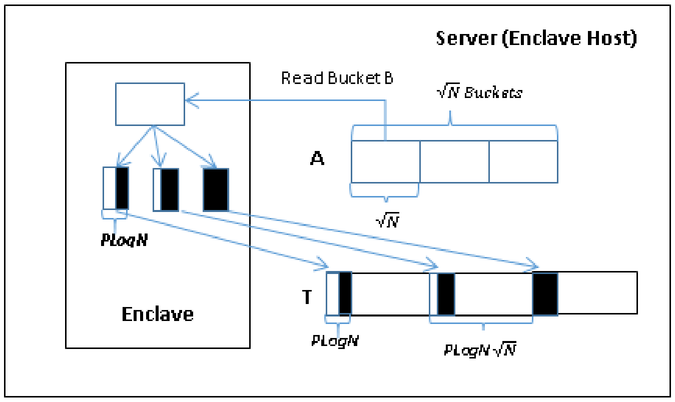

Fig. 8: Illustration of the first step in Melbourne shuffle in SGX's SQRT ORAM. The shaded parts are dummies. Some of the T blocks with the size of $p \log N$ (such as the third block) are all dummies which means that no items are permuted to the corresponding block in T. In this example, we assume that $\sqrt{N}$=3. Accordingly, we have 3 buckets in A of size $\sqrt{N}$ each and 3 buckets in T of size $p \log N \sqrt{N}$ each.

$p \log N$ is most probably larger than the number of items which belong to each block, so we fill the rest of the block with dummy items. The locations of the dummies in a block are chosen randomly. Fig. 8 illustrates this stage.
Algorithm 6 is the algorithm used to perform this stage. In order to find the elements in bucket B which belong to a specific output block, one can naively scan B. However, since we have $\sqrt{N}$ output blocks for each bucket B, we would need N steps for every bucket B. Accordingly, we would need O($N\sqrt{N} \log N$) time for the whole stage. This problem may not exist in a traditional Melbourne shuffle (non-SGX client-server setting) since it is expected that the client, on the contrary to an enclave, can handle one block of size $\sqrt{N}p \log N$ items. To tackle this issue, we use the following technique for each bucket B in A:

  - Three arrays are needed: $counters$, $counterAccs$, and $ptrs$ of size $\sqrt{N}$ each.
  - We scan bucket B once. We store the counts of elements which belong to each output block in $counters$ so that $counters[i]$ has the number of elements which should be copied into the i-th output block.
  - We set $counterAccs[1] = counters[1]$. For $2 \leq i \leq \sqrt{N}$:
    we set $counterAccs[i] = counters[i-1] + counters[i]$. $counterAccs[i]$ contains the number of elements which should be copied to all output blocks up to the i-th output block (accumulated counter).
  - Since $counters$ and $counterAccs$ are filled, it is easy to fill $ptrs$ array by scanning B again. $ptrs$ will ultimately have pointers to all elements in B ($ptrs[1..counters[1]]$ has pointers

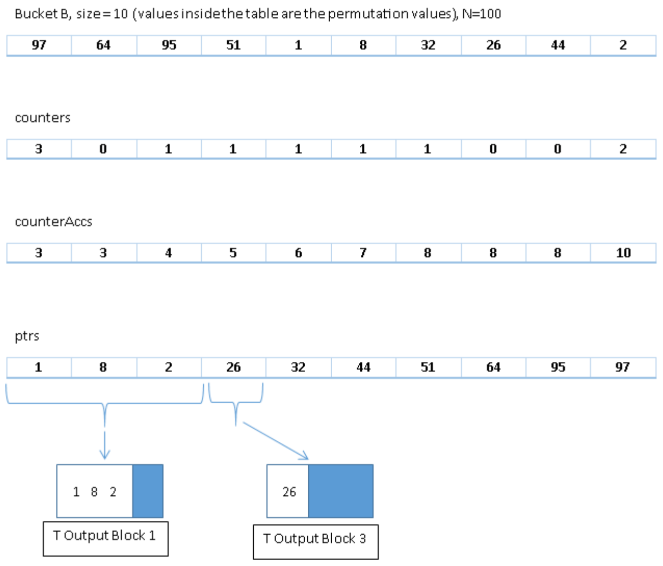

Fig. 9: An example showing the procedure of filling T out of Bucket B. We assume that this is the first bucket in A. Note that output block 2 which is missing has only dummies. This procedure is repeated $\sqrt{N}$ times since we have $\sqrt{N}$ buckets in A of size $\sqrt{N}$ each. $ptrs$ shows the permutation values of the items to which it is pointing.

to items which are permuted to the first output block). To fill an output block, we know how many elements to fill and where these elements are by utilizing our three arrays. Fig. 9 shows an example for processing a bucket of size 10 where N=100. First element in array $counters$ is 3 since we have three 3 elements in B permuted to the first output block in T.

With this technique, stage 1 requires only O($N \log N$) processing time. However, it requires an extra space of 3 $\times \sqrt{N}$ short integers (not items), assuming that $\sqrt{N} < 2^{16}$. Function $perm(i.pos)$ returns the permutation value of the original index of item i. The reason for storing the original index of block i is its necessity in the re-initialization process. When data are brought back to the enclave during the re-initialization, the original indices are lost unless we can inverse the permuted value. Line 5 is required since the Melbourne shuffle is handling $N + \sqrt{N}$ items. obv_bk is an array of two items. One item is used to process the "real" item and the other for processing the dummies. To prevent an adversary from knowing the number of items in each T block, loops in lines (23-37) use $obv\_bk[1]$ to include the real item and $obv\_bk[0]$ to include the dummy.

- *Stage 2: Filling array O out of T.* Since T is filled, we can move to stage 2 which is to read each bucket of $\sqrt{N} p \log N$ items from T and fill each corresponding bucket of $\sqrt{N}$ items of the final output array O. It is mandatory to fill each bucket of $\sqrt{N}$ items in O in one step to achieve obliviousness, otherwise, an adversary would be able to find out the number of items in each block (of size $pLogN$) in T. Accordingly, an array of size $\sqrt{N}$ should be kept inside the enclave to be sent after processing to O. However, a bucket of size $\sqrt{N} p \log N$ from T may be too big to fit in our enclave, so we modify the original algorithm to read it in blocks of size $p \log N$. Fig. 10 explains stage 2. For each block, dummy items are removed and the "real" items are copied to their locations in O. Since stage 2 is straightforward, no pseudo code is provided.

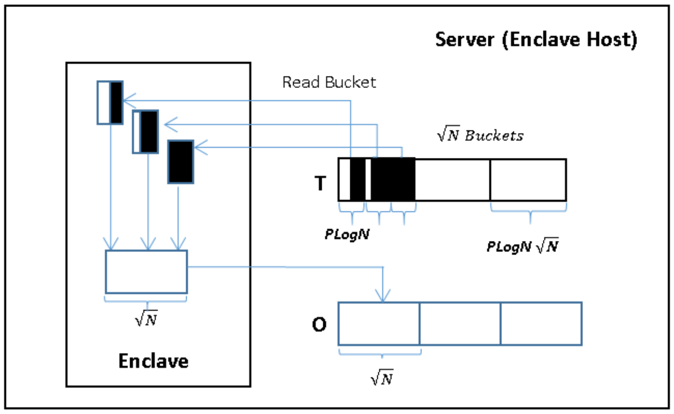

Fig. 10: Illustration of the second step in Melbourne shuffle in SGX SQRT ORAM. Blocks of size $p \log N$ are read sequentially into the enclave and dummies are removed. When a bucket of $\sqrt{N}$ items is filled inside the enclave, it is encrypted and copied to O.