# PRO-ORAM: Practical Read-Only Oblivious RAM

Shruti Tople* *Microsoft Research*
Yaoqi Jia *Zilliqa Research*
Prateek Saxena *NUS*

## Abstract

Oblivious RAM is a well-known cryptographic primitive to hide data access patterns. However, the best known ORAM schemes require a logarithmic computation time in the general case which makes it infeasible for use in real-world applications. In practice, hiding data access patterns should incur a constant latency per access.

In this work, we present PRO-ORAM— an ORAM construction that achieves *constant latencies* per access in a large class of applications. PRO-ORAM theoretically and empirically guarantees this for *read-only* data access patterns, wherein data is written once followed by read requests. It makes hiding data access pattern practical for read-only workloads, incurring sub-second computational latencies per access for data blocks of 256 KB, over large (gigabyte-sized) datasets. PRO-ORAM supports throughputs of tens to hundreds of MBps for fetching blocks, which exceeds network bandwidth available to average users today. Our experiments suggest that dominant factor in latency offered by PRO-ORAM is the inherent network throughput of transferring final blocks, rather than the computational latencies of the protocol. At its heart, PRO-ORAM utilizes key observations enabling an aggressively parallelized algorithm of an ORAM construction and a permutation operation, as well as the use of trusted computing technique (SGX) that not only provides safety but also offers the advantage of lowering communication costs.

## 1 Introduction

Cloud storage services such as Dropbox [4], Google Drive [8], Box [2] are becoming popular with millions of users uploading Gigabytes of data everyday [6]. However, outsourcing data to untrusted cloud storage poses several privacy and security issues [5]. Although encryption of data on the cloud guarantees data confidentiality, it is not sufficient to protect user privacy. Research has shown that access patterns on encrypted data leak substantial private information such as secret keys and user queries [25, 27]. One line of research to stop such inference is the use of Oblivious RAM (ORAM) [22]. ORAM protocols continuously shuffle the encrypted data blocks to avoid information leakage via the data access patterns.

Although a long line of research has improved the performance overhead of ORAM solutions [20, 32, 37, 40, 42, 43], it is still considerably high for use in practice. Even the most efficient ORAM solutions incur at least logarithmic latency to hide read / write access patterns [20, 34, 43], which is the established lower bound for the general case. Ideally, hiding access patterns should incur a *constant* access (communication) latency for the client, independent of the size of data stored on the cloud server, and constant computation time per access for the cloud server. To reduce the logarithmic access time to a constant, we investigate the problem of designing solutions to hide specific patterns instead of the general case.

We observe that a large number of cloud-based storage services have a read-only model of data consumption. An application can be categorized in this model when it offers only read operations after the initial upload (write) of the content to the cloud. For example, services hosting photos (e.g., Flickr, Google Photos, Moments), music (e.g., Itunes, Spotify), videos (e.g., NetFlix, Youtube) and PDF documents (e.g., Dropbox, Google Drive) often exhibit such patterns. Recently, Blass et al. have shown that designing an efficient construction is possible for "write-only" patterns wherein the read accesses are not observable to the adversary (e.g. in logging or snapshot / sync cloud services) [18]. Inspired by such specialized solutions, we ask *whether it is possible to achieve constant latency to hide read-only access patterns?* As our main contribution, we answer the above question affirmatively for all cloud-based data hosting applications.

### 1.1 Approach

We propose PRO-ORAM— a practical ORAM construction for cloud-based data hosting services offering constant latency for read-only accesses. PRO-ORAM incurs a constant computation and communication latency per access making it a promising

*Work done as a Ph.D student at National University of Singapore (NUS)

solution to use in a large class of real-world applications. The key idea to achieve constant latencies is to decompose every request to read a data block into two separate sub-tasks of "access" and "shuffle" which can execute in parallel. However, simply parallelizing the access and shuffle operations is not enough to achieve constant latencies. Previous work that employs such parallelization for the general case would incur a logarithmic slowdown even for read-only accesses due to the inherent design of the underlying ORAM protocols [41].

In designing PRO-ORAM, we make two important observations that allow us to achieve constant latency. First, we observe that there exists a simple ORAM construction — the square-root ORAM [22] — which can be coupled with a secure permutation (or shuffle) [33] to achieve idealized efficiency in the read-only model. A naïve use of this ORAM construction incurs a worst-case overhead of $O(N \log^2 N)$ to shuffle the entire memory with $N$ data blocks. The non-updatable nature of read-only data allows us to parallelize the access and shuffle operations on two separate copies of the data. This results in a de-amortized $O(\sqrt{N})$ latency per access.

Second, we design a secure method to distribute the work done in each shuffle step among multiple computational units without compromising the original security guarantees. Our construction still performs $O(\sqrt{N})$ work per access but it is parallelized aggressively to execute in a constant time. Assuming a sufficient number of cores, PRO-ORAM distributes the total shuffling work among $O(\sqrt{N})$ threads without leaking any information. Although the total computation work is the same as in the original shuffle algorithm, the latency reduces to a constant for read streaks[1]. With these two observations, we eliminate the expensive $O(N \log^2 N)$ operation from stalling subsequent read access requests in PRO-ORAM. Thus, we show that a basic ORAM construction is better for hiding read data access patterns than a complex algorithm that is optimized to handle the general case. Further, we present a proof for the correctness and security of PRO-ORAM. Our improved construction of the shuffle algorithm maybe of independent interest, as it is widely applicable beyond ORAM.

PRO-ORAM can be applied opportunistically for applications that expect to perform long streaks of read accesses intermixed with infrequent writes, incurring a non-constant cost only on write requests. Therefore, PRO-ORAM extends obliviousness to the case of arbitrary access patterns, providing idealized efficiency for "read-heavy" access patterns (where long streaks of reads dominate). To reduce trust on software, PRO-ORAM assumes the presence of a trusted hardware (such as Intel SGX [1], Sanctum [19]) or a trusted proxy as assumed in previous work on ORAMs [16, 28, 41].

### 1.2 Results

We implement PRO-ORAM prototype in C/C++ using Intel SGX Linux SDK v1.8 containing 4184 lines of code [9]. We evaluate PRO-ORAM using Intel SGX simulator for varying file / block sizes and total data sizes. Our experimental results demonstrate that the latency per access observed by the user is a *constant* of about 0.3 seconds to fetch a file (or block) of size 256 KB. Our empirical results show that PRO-ORAM is practical to use with a throughput ranging from 83 Mbps for block size of 100 KB to 235 Mbps for block size of 10 MB. These results are achieved on a server with 40 cores. In real cloud deployments, the cost of a deca-core server is about a thousand dollars [10]; so, the one-time setup cost of buying 40 cores worth of computation seems reasonable. Thus, PRO-ORAM is ideal for sharing and accessing media files (e.g., photos, videos, music) having sizes of few hundred KB on today's cloud platforms. PRO-ORAM's throughput exceeds the global average network bandwidth of 7 Mbps asserting that the inherent network latency dominates the overall access time rather than computation latencies in PRO-ORAM [7].

**Contributions.** We summarize our contributions below:

- *Read-only ORAM*. We present PRO-ORAM— a practical and secure read-only ORAM design for cloud-based data hosting services. PRO-ORAM's design utilizes sufficient computing units equipped with a trusted hardware primitive.
- *Security Proof*. We provide a security proof to guarantee that our PRO-ORAM construction provides obliviousness in the read-only data model.
- *Efficiency Evaluation*. PRO-ORAM is highly practical with constant latency per access for fixed block sizes and provides throughput ranging from 83 Mbps for a block size of 100 KB to 235 Mbps for a block size of 10 MB.

## 2 Overview

Our main goal is to ensure two important characteristics: a) hide read data access patterns on the cloud server; and b) achieve constant time to access each block from the cloud.

### 2.1 Setting: Read-Only Cloud Services

Many applications offer data hosting services for images (e.g., Flickr, Google Photos, Moments), music (e.g., Itunes, Spotify), videos (e.g., NetFlix, Youtube), and PDF documents (e.g., Dropbox, Google Drive). In these applications, either the users (in the case of Dropbox) or the service providers (such as NetFlix, Spotify) upload their data to the cloud server. Note that the cloud provider can be different from the service provider, for example, Netflix uses Amazon servers to host their data. After the initial data is uploaded, users mainly perform read requests to access the data from the cloud.

Let a data owner upload $N$ files each having a *file identifier* to the cloud. A file is divided into *data blocks* of size $B$

[1] A read streak is a sequence of consecutive read operations.

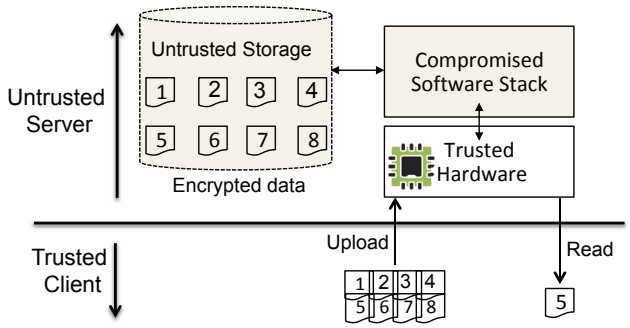



Figure 1: Baseline setting: Cloud-provider with trusted hardware and a compromised software stack. User uploads data and makes read requests

and stored in an *array* on the untrusted storage at the server. Each block is accessed using its corresponding *address* in the storage array. To handle variable length files, one can split large files into several data blocks and maintain a file to blocks mapping table. However, for simplicity, we assume each file maps to a single block and hence use the terms file and block interchangeably in this paper. When a user requests to fetch a file, the corresponding data block is read from the storage array and is sent to the user. To ensure confidentiality of the data, all the files are encrypted using a cryptographic key. The data is decrypted only on the user machine using the corresponding key.

## 2.2 Threat Model

Leakage of access patterns is a serious issue and has been shown to leak critical private information in several settings such as encrypted emails, databases and others [25, 27]. In our threat model, we consider that the adversary has complete access to the encrypted storage on the cloud. An attacker can exploit the vulnerabilities in the cloud software to gain access to the cloud infrastructure including the storage system which hosts encrypted content [5, 12]. Hence, we consider the cloud provider to be untrusted with a compromised software stack. The cloud provider can trace the requests or file access patterns of all the users accessing the encrypted data. We restrict each request to only read the data from the server. Essentially, the adversary can observe the exact address accessed in the storage array to serve each requested file. Along with access to the storage system, the adversary can observe the network traffic consisting of requested data blocks sent to each user.

**Scope.** Our main security goal is to guarantee obliviousness i.e., hide read access patterns of users from the cloud provider. Although we consider a compromised server, we do not defend against a cloud provider refusing to relay the requests to the user. Such denial of service attacks are not within the scope of this work. We only focus on leakage through *address* access patterns and do not block other channels of leakage such as timing or file length [44]. For example, an adversary can observe the number of blocks fetched per request or the frequency of requesting files to glean private information about the user. However, our system can benefit from existing solutions that thwart these channels using techniques such as padding files with dummy blocks and allowing file requests at fixed interval respectively [15].

## 2.3 Baseline: Trusted H/W in the Cloud

A well-known technique to hide data access patterns is using Oblivious RAM (ORAM) [22]. In ORAM protocols, the encrypted data blocks are obliviously shuffled at random to unlink subsequent accesses to the same data blocks. Standard ORAM solutions guarantee obliviousness in a trusted client and an untrusted server setting. It generally uses a private memory called stash at the client-side to perform oblivious shuffling and re-encryption of the encrypted data. In the best case, this results in a logarithmic communication overhead between the client and the server [43]. To reduce this overhead, previous work has proposed the use of a trusted hardware / secure processor [16, 28] in the cloud or a trusted proxy [41]. This allows us to establish the private stash and a small trusted code base (TCB) to execute the ORAM protocol in the cloud. That is, instead of the client, the trusted component on the cloud shuffles the encrypted data, thereby reducing the communication overhead to a constant. Further, the trusted component can verify the integrity of the accessed data and protect against a malicious cloud provider [41]. Figure 1 shows the architecture for our baseline setting with a trusted hardware and a compromised software stack on the cloud.

In this work, we consider the above cloud setup with a trusted hardware as our baseline. Specifically, we assume the cloud servers are equipped with Intel SGX-enabled CPUs. SGX allows creating hardware-isolated memory region called *enclaves* in presence of a compromised operating system. With enclaves, we have a moderate size of private storage inaccessible to the untrusted software on the cloud. Further, we assume that the trusted hardware at the cloud provider is untampered and all the guarantees of SGX are preserved. We do not consider physical or side-channel attacks on the trusted hardware [26, 29, 31, 38, 45]. Defending against these attacks is out of scope but our system can leverage any security enhancements available in the future implementation of SGX CPUs [30]. In practice, SGX can be replaced with any other trusted hardware primitive available in the next-generation cloud servers.

## 2.4 Solution Overview

We present a construction called `PRO-ORAM`— a Practical Read-Only ORAM scheme that achieves *constant computation* latencies for read streaks. `PRO-ORAM` is based on square-root ORAM but can be extended by future work to other

ORAM approaches. It incurs default latency of the square-root ORAM approach in case of write operations. Thus, one can think of PRO-ORAM as a specialization for read streaks, promising most efficiency in applications that are read-heavy, but without losing compatibility in the general case.

**Key Insight 1.** The dominant cost in any ORAM scheme comes from the shuffling step. In square-root ORAM, the shuffling step is strictly performed after the access step [22]. This allows the shuffle step to consider any updates to the blocks from write operations. Our *main* observation is that for read-only applications, the algorithm need not wait for all the accesses to finish before shuffling the entire dataset. The key advantage in the read-only model is that the data is never modified. Thus, we can *decouple* the shuffling step from the logic to dispatch an access. This means the shuffle step can execute *in parallel* without stalling the read accesses. We give a proof for the correctness and security of PRO-ORAM in Section 5. Although prior work has considered parallelizing the access and shuffle step [41], our observations only apply to the read-only setting, and our specific way achieves constant latency which was not possible before.

**Key Insight 2.** Our second important observation allows us to reach our goal of constant latency. We observe that the Melbourne Shuffle algorithm performs $O(\sqrt{N})$ computation operations for each access where each operation can be executed independently [33]. Hence, the $O(\sqrt{N})$ computations can be performed in parallel (multi-threaded) without breaking any security or functionality of the original shuffle algorithm. This final step provides us with a highly optimized Melbourne Shuffle scheme which when coupled with square-root ORAM incurs constant computation latency per access. We further exploit the structure of the algorithm and propose pipelining based optimizations to improve performance by a constant factor (Section 4.4). We remark that our efficient version of the shuffle algorithm maybe of independent interest and useful in other applications [21, 33].

Note that PRO-ORAM is *compatible* with data access patterns that have writes after read streaks, since it can default to running a synchronous (non-parallel) shuffle when a write is encountered — just as in the original square-root ORAM. Of course, the constant latency holds for read streaks and read-heavy applications benefit from this specialized construction.

**Comparison to Previous Work.** The most closely related work with respect to our trust assumptions and cloud infrastructure is ObliviStore [41]. This protocol has the fastest performance among all other ORAM protocols when used in the cloud setting [17]. Similar to PRO-ORAM, ObliviStore parallelizes the access and shuffle operations using a trusted proxy for cloud-based data storage services.

We investigate whether ObliviStore's construction can attain constant latency when adapted to the read-only model. We highlight that although the high-level idea of parallelizing ORAM protocol is similar to ours, ObliviStore differs from PRO-ORAM in various aspects. ObliviStore is designed to hide arbitrary patterns in the general case and hence uses a complex ORAM protocol that is optimized for bandwidth. It uses a partition-based ORAM [42] where each partition is itself a hierarchical ORAM [22]. This design takes O(log N) time to access each block even if the protocol was restricted to serve read-only requests. Hence, our observations in the read-only model do not directly provide performance benefits to ObliviStore's construction. The key factor in PRO-ORAM is that — "simple and specialized is better" — a simple ORAM construction which is non-optimized for the general case, is better suited for hiding read-only data access patterns.

## 3 Background

In designing an efficient PRO-ORAM scheme, we select square-root ORAM as our underlying ORAM scheme as it allows $\sqrt{N}$ accesses before the shuffling step. To obliviously shuffle the data in parallel with the accesses, we select the Melbourne shuffle scheme, that allows shuffling of data of O(N) in $O(\sqrt{N})$ steps. Further, we use Intel SGX-enabled CPU present to create enclaves with $O(\sqrt{N})$ private storage. We provide a brief background on each of these building blocks.

### 3.1 Square-Root ORAM

We select the square-root ORAM scheme as the underlying building block in PRO-ORAM. The square-root ORAM scheme, as proposed by Goldreich and Ostrovsky [22], uses $N+\sqrt{N}$ permuted memory and a $\sqrt{N}$ *stash* memory, both of them are stored encrypted on the untrusted cloud storage. The permuted memory contains $N$ real blocks and $\sqrt{N}$ dummy blocks arranged according to a pseudo-random permutation $\pi$.

To access a block, the protocol first scans the entire stash deterministically for the block. If the requested block is found in the stash then the protocol makes a fake access to a dummy block in the permuted memory. Otherwise, it accesses the real block from the permuted memory. The accessed block is then written to the stash by re-encrypting the entire $\sqrt{N}$ stash memory. The key trick here is that all accesses exhibit a deterministic access order to the adversarial server, namely: a deterministic scan of the stash elements, followed by an access to a real or dummy block in permuted memory, followed by a final re-encrypted write and update to the stash. After every $\sqrt{N}$ requests, the protocol updates the permuted memory with the stash values and obliviously permutes (shuffles) it randomly. This shuffling step incurs $O(N\log^2 N)$ overhead, resulting in an amortized latency of $O(\sqrt{N}\log^2 N)$ per request.

### 3.2 Intel SGX

Recently, Intel proposed support for a trusted hardware primitive called Software Guard Extensions (SGX). With SGX, we can create isolated memory regions called *enclaves* which are

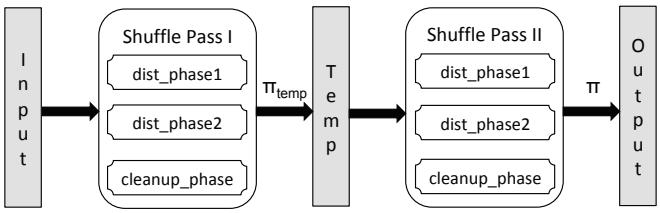



Figure 2: Overview of the Melbourne shuffle algorithm

inaccessible to the underlying operating system or any other application. In PRO-ORAM, we use the following two important features of Intel SGX. PRO-ORAM can be build using any other trusted hardware that provides these specific features.

**Enclaved Memory.** SGX allows the creation of hardware-isolated private memory region or enclaved memory. For SGX CPUs, BIOS allocates a certain region for processor reserved memory (PRM) at the time of boot up. The underlying CPU reserves a part of this PRM to create enclaves. All the code and data in the enclaved memory is inaccessible even to the privileged software such as the OS. Thus, an adversary in our threat model cannot access this protected memory. It guarantees confidentiality of the private data within enclaves from the adversary. At present, SGX supports 90 MB of enclaved memory. This allows us to use a moderate amount of private storage at the cloud provider. Further, we can create multiple threads within an enclave [39].

**Attestation.** Along with enclaved execution, SGX-enabled CPUs support remote attestation of the software executing within an enclave. This security features enables a remote party to verify the integrity of the software executing on an untrusted platform such as the cloud. Further, it supports local attestation between two enclaves executing on the same machine. These enclaves can then establish a secure channel and communicate with each other. One can perform such attestation of an enclave program as described in the SGX manual [1]. Thus, SGX-enabled CPUs at the cloud provider allows executing trusted code base (TCB) with a small amount of private storage at the cloud provider.

## 3.3 Melbourne Shuffle

Melbourne shuffle is a simple and efficient randomized oblivious shuffle algorithm [33]. Using this algorithm, we can obliviously shuffle $N$ data blocks with $O(N)$ external memory. The data is stored at the server according to a pseudo-random permutation. The encryption key and the permutation key $\pi$ require constant storage and are stored in the private memory. This algorithm uses private storage of the size $O(\sqrt{N})$ and incurs a communication and message complexity of $O(\sqrt{N})$. We use this algorithm in PRO-ORAM to shuffle the encrypted data in parallel to accessing data blocks using enclave memory as the private storage.

The algorithm works in two passes as shown in Figure 2. It first shuffles the given input according to a random permutation $\pi_{\texttt{temp}}$ and then shuffles the intermediate permutation to the desired permutation of $\pi$. Each pass of the shuffle algorithm has three phases, two distribution and a cleanup phase. The algorithm divides each $N$ size array into buckets of size $\sqrt{N}$. Further, every $\sqrt[4]{N}$ of these buckets are put together to form a chunk. Thus, the $N$ array is divided into total $\sqrt[4]{N}$ chunks. The first distribution phase (dist_phase1) simply puts the data blocks into correct chunks based on the desired permutation $\pi_{\texttt{temp}}$ in the first pass and $\pi$ in the second pass. The second distribution phase (dist_phase2) is responsible for placing the data blocks into correct buckets within each chunk. Finally, the clean up phase (cleanup_phase) arranges the data blocks in each bucket and places them in their correct positions based on the permutation key.

Choosing appropriate constants in the algorithm guarantees oblivious shuffling of $N$ data blocks for any chosen permutation value $\pi$ with a very high probability. The important point is that each of these phases can be implemented to have a "constant" depth and operate "independently" based only on the pre-decided $\pi_{\texttt{temp}}$ and $\pi$ values. This allows us to distribute the overall computation among multiple threads and parallelize the algorithm. Although the total work done remains the same, our design effectively reduces the overall execution time to a constant. We refer readers to the original paper for the detailed algorithm of each of these phases [33].

## 3.4 Encryption Algorithms

We use standard symmetric key and public key cryptographic schemes as our building blocks in PRO-ORAM. We assume that both these schemes guarantee IND-CPA security. The security guarantees of PRO-ORAM depends on the assumption of using secure underlying cryptographic schemes. We denote by $SE = (Gen_{SE}, Enc_{SE}, Dec_{SE})$ a symmetric key encryption scheme where $Gen_{SE}$ algorithm generates a key which is used by the $Enc_{SE}$ and $Dec_{SE}$ algorithms to perform encryption and decryption respectively. $PKE = (Gen_{PKE}, Enc_{PKE}, Dec_{PKE})$ denotes a public key encryption scheme where the $Gen_{PKE}$ algorithm generates a public-private key pair $(Pb, Pr)$. The $Enc_{PKE}$ algorithm takes the public key $Pb$ as input and encrypts the data whereas the $Dec_{PKE}$ takes the private key $Pr$ as input and decrypts the ciphertext.

# 4 PRO-ORAM Details

Today's cloud platforms are equipped with a large amount of storage and computing units. In PRO-ORAM, we leverage these resources to achieve practical performance guarantees for hiding access patterns to read-only data such as photos, music, videos and so on.

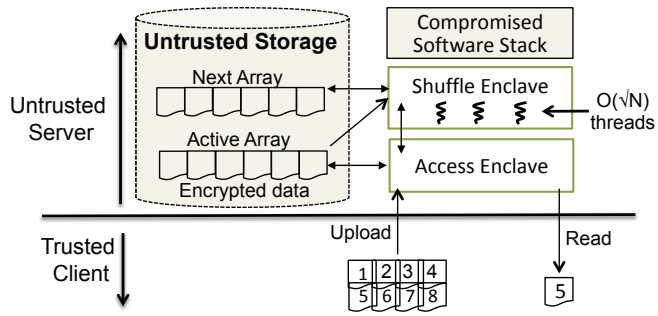



Figure 3: PRO-ORAM design overview with access and shuffle enclaves operating in paralllel on active and next array.

## 4.1 Design Overview

Similar to any cloud storage service, we have a setup phase to establish user identities and upload initial data to the cloud. We outline the setup phase for users that directly upload their data to the cloud storage for e.g., Dropbox or Google Drive. However, it can be modified to accommodate applications such NetFlix, Spotify where the initial data is uploaded by the service providers and not the users themselves.

**Initialization.** Each user registers with the cloud provider his identity uid and a public key $\mathrm{Pb}_{\mathrm{uid}}$ mapped to their identity. Let the data structure Pub_map store this mapping on the server. The private key $\mathrm{Pr}_{\mathrm{uid}}$ corresponding to the public key is retained by the user. Each of these registered users can upload their data to the server. To upload $N$ data blocks to the untrusted server, a data owner first encrypts the data blocks with a symmetric key $K$ and then sends them to the server. The order of these blocks during the initial upload does not affect the security guarantees of PRO-ORAM. On receiving the encrypted data, the server instantiates an "access" and a "shuffle" enclave. Next, the data owner attests the program running within these enclaves and secretly provisions the encryption key $K$ to them on successful attestation.

**System Overview.** Figure 3 shows the overview of PRO-ORAM design for the read-only model. PRO-ORAM executes two enclaves called access and shuffle in parallel on the untrusted server. Each access and shuffle enclave has $O(\sqrt{N})$ private storage and corresponds to a set of $N$ data blocks. These enclaves provide obliviousness guarantees to read from the $N$ data blocks uploaded on the server. The enclaves locally attest each other and establish a secure channel between them [13]. They communicate over the secure channel to exchange secret information such as encryption and permutation keys (explained in detail in Section 4.2). The access enclave executes the square-root ORAM and the shuffle enclave performs the Melbourne shuffle algorithm. However, PRO-ORAM parallelizes the functioning of both these enclaves to achieve constant latency per read request.

**Algorithm 1:** Pseudocode for each round of shuffle enclave

**Input:** active_array: input data blocks ,
$\mathrm{K}_{\mathrm{prev}}$: previous key,
$\mathrm{K}_{\mathrm{new}}$: new key,
$\pi$: desired permutation,
r_num: current round number
**Output:** next_array: output permuted blocks

```
1  Let T1, T2, O_temp be temporary arrays;
2  Let π_temp be a random permutation;
3  Let K_temp be a encryption key;
4  if r_num == 0 then
       // Add dummy blocks
5      for j from N to N + √N do
6          d'_j ← Enc_SE( K_prev, dummy);
7          active_array = active_array ∪ d'_j;
8      end
9  end
   // Two pass call to shuffle algorithm
10 mel_shuffle(active_array, T1, T2, π_temp, K_prev, K_temp,
     O_temp);
11 mel_shuffle(O_temp, T1, T2, π, K_temp, K_new, next_array);
```

PRO-ORAM algorithm consists of several rounds where each round is made of total $\sqrt{N}$ requests from the users. In every round, the access enclave strictly operates on the permuted array of the uploaded data, which we refer as the active array. On every request, the access enclave fetches the requested data block either from the active array or the private stash (similar to the square-root ORAM), re-encrypts the block and sends it to the user. Simultaneously, the shuffle enclave reads data blocks in a deterministic pattern from the active array, performs the shuffle algorithm on them and outputs a new permuted array, which we refer as the next array. The shuffle enclave internally distributes the work using $O(\sqrt{N})$ separate threads. By the end of each round, i.e., after $\sqrt{N}$ requests, the active array is replaced with the next array. Thus, for serving $N$ data blocks, PRO-ORAM uses $O(N)$ space on the server to store the active and the next array.

Parallelizing the access and shuffle enclave enables PRO-ORAM to create a new permuted array while serving requests on the active array. This design is novel to PRO-ORAM and differs from previous ways of parallelizing access and shuffle operations [23, 41]. The algorithms for both the access and shuffle operations execute within SGX enclaves and are oblivious to the server. We give a detailed proof in Section 5.

## 4.2 Shuffle Enclave

The shuffle enclave starts its execution one round before the access enclave. We call this as the preparation round or round 0. The shuffle enclave uses this round to per-

**Algorithm 2:** Parallel pseudocode for `mel_shuffle` function

**Input:** I: input data blocks ,
$T_1$, $T_2$: Temporary arrays,
$K_{prev}$: previous key,
$K_{new}$: new key,
$\pi$: desired permutation,
**Output:** O: output permuted blocks

1 Let $K_1$, $K_2$ be encryption keys;
// Place the blocks into correct chunks
2 dist_phase1(I, $\pi$, $K_{prev}$, $K_1$,$T_1$):: $O(\sqrt{N})$ threads;
// Place the blocks in correct buckets
3 dist_phase2($T_1$, $\pi$, $K_1$, $K_2$, $T_2$):: $O(\sqrt{N})$ threads;
// Arrange the blocks in each bucket
4 cleanup_phase($T_2$, $\pi$, $K_2$, $K_{new}$):: $O(\sqrt{N})$ threads;

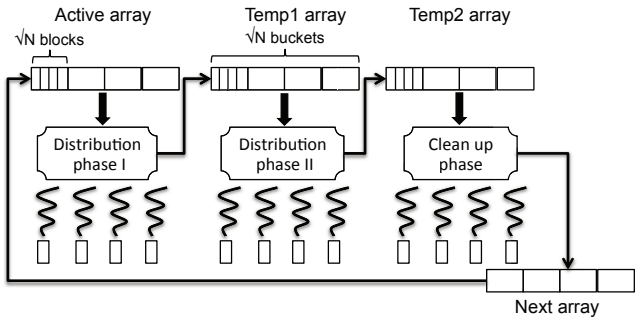



Figure 4: Multi-threaded Melbourne shuffle with constant latency per access

mute the data which is uploaded by the user during the initialization phase. The enclave permutes $N$ encrypted real blocks $(d'_1, \cdots, d'_N)$ along with $\sqrt{N}$ dummy blocks and adds them to the `active` array (as shown in lines 4-9 in Algorithm 1). In each round, the enclave executes the Melbourne shuffle algorithm with the `active` array as input and the `next` array as output. It makes a two pass call to the `mel_shuffle` function (lines 10 and 11). Internally, the function performs the three phases of `dist_phase1`, `dist_phase2` and `clean_up_phase` (lines 2, 3, 4 in Algorithm 2). Each phase performs $\sqrt{N}$ steps, where each step fetches $\sqrt{N}$ blocks of the input array, re-arranges and re-encrypts them and writes to the output array.

In `PRO-ORAM`, we distribute this computation over $O(\sqrt{N})$ threads and thus parallelize the execution of each phase (as shown in Figure 4). Carefully selecting the hidden constants in $O(\sqrt{N})$ allows us to securely distribute the work without compromising on the security of the original algorithm (see Lemma 5.1 in Section 5). Each thread re-encrypts and re-arranges only a single block in every step of the phase and writes them back in a deterministic manner. The operations on each block are independent of other blocks and

**Algorithm 3:** Pseudocode for `Read` Algorithm

**Input:** $d_i$: block identifier,
active_array: encrypted data blocks,
request: current request number
**Output:** d': encrypted block

1 Lookup in the private stash;
2 **if** $d_i$ **in** `stash` **then**
  // access dummy value
3  addr $\leftarrow \pi(N + \texttt{request})$;
4  d' $\leftarrow$ active_array(addr) ;
  // select value from stash
5  d' $\leftarrow$ stash($d_i$);
6 **else**
7  addr $\leftarrow \pi(d_i)$ ;
8  d' $\leftarrow$ active_array(addr) ;
9  Write d' to the stash;
10 **end**
11 **return** $d'$;

have a constant depth. The threads use the private memory within the enclave as a stash to obliviously shuffle the blocks. However, each thread reads and writes to its corresponding memory location during the shuffling step. We exploit this property and parallelize the computation on each of these blocks. In `PRO-ORAM`, we implement this approach using multi-threading with SGX enclaves. The `shuffle` enclave starts $O(\sqrt{N})$ threads in parallel to compute the re-encryption and rearrangement of data blocks. This results in a constant computation time per step. Thus, with parallelization imposed in each step, the total computation time for shuffling $N$ data blocks is $O(\sqrt{N})$. Hence, the amortized computation latency per request over $\sqrt{N}$ requests is reduced to $O(1)$. `PRO-ORAM` distributes the work in each shuffle step over $O(\sqrt{N})$ threads.

After the shuffle is completed, the next_array is copied to the active_array. The `shuffle` enclave sends the new keys ($K_{new}$) and permutation value ($\pi$) to the `access` enclave using a secure channel established initially. The latter enclave uses these keys to access the correct requested blocks from the active_array in the next round.

## 4.3 Access Enclave

Unlike the `shuffle` enclave, the `access` enclave begins execution from round 1. Each round accepts $\sqrt{N}$ read requests from the users. Before the start of each round, the `access` enclave gets the permutation $\pi$ and encryption key $K_{new}$ from the `shuffle` enclave. The `active` array corresponds to data blocks shuffled and encrypted using the keys $\pi$ and $K_{new}$. For each request, the `access` enclave takes as input the block identifier $d_i$ and the requesting user id `uid`. The enclave first confirms that the requesting `uid` is a valid registered user and

**Algorithm 4:** Pseudocode for each round of `access` enclave

**Input:** $d_i$: request file identifier ,
Pub_map: User id and public key mapping table ,
uid: requesting user id,
$K_{new}$: encryption key
$\pi$: permutation key
active_array: permuted array
**Output:** response_msg

```
1 for request from 1 to √N do
2 |   Pb_uid ← Pub_map (uid);
3 |   d' ← Read (d_i, active_array, request);
4 |   k' ← Gen_SE;
5 |   d'' ← Enc_SE(Dec_SE(d', K_new), k');
6 |   key_msg = Enc_PKE(Pb_uid, k'');
7 |   response_msg = (d'', key_msg);
8 end
```

has a public key corresponding to the user. On confirmation, the enclave invokes the read function in Algorithm 3.

The algorithm to read a block is same as the main logic of square-root ORAM. Algorithm 3 provides the pseudocode for reading a data block in PRO-ORAM. Note that, we do not store the private stash on the untrusted cloud storage as proposed in the original square-root ORAM approach. Instead, the stash is maintained in the private memory within the access enclave. The stash is indexed using a hash table and hence can be looked up in a constant time. The read algorithm checks if the requested data block is present in the private stash. If present, the enclave accesses a dummy block from the untrusted storage. Else, it gets the address for the requested block $d_i$ using permutation $\pi$ and fetches the real block from the untrusted storage. The output of the read algorithm is an encrypted block $d'$. The block is stored in the private stash.

After fetching the encrypted block $d'$, either from the private stash or the `active` array, the `access` enclave decrypts it using $K_{new}$. Algorithm 4 shows the pseudocode for this step. It then selects a new key $k'$ and encrypts the block. The output message includes this re-encrypted block and the encryption of $k'$ under public key of the requesting user $Pb_{uid}$. At the end of each round i.e., after serving $\sqrt{N}$ request, the access enclave clears the private storage, permutation $\pi$ and $K_{new}$. Note that unlike the original square-root ORAM, there is no shuffling after $\sqrt{N}$ requests. The permuted `next` array from the `shuffle` enclave replaces the `active` array.

**Performance Analysis.** In PRO-ORAM, the `access` enclave sends only the requested block to the user. This results in a communication overhead of $O(1)$ with respect to the requested block size. Further, the `access` enclave computes i.e., re-encrypts only a single block for each request. Thus, the computation on the server for the `access` enclave is $O(1)$. The `shuffle` enclave computes a permuted array in $O(\sqrt{N})$ steps. It fetches $O(\sqrt{N})$ blocks for each request. Note that the total computation performed at the server is still $O(\sqrt{N})$ for each request. However, in PRO-ORAM, we parallelize the computation i.e, re-encryption on $O(\sqrt{N})$ blocks in $O(\sqrt{N})$ threads. This reduces the computation time required for each step to only a single block. Thus, the overall computation latency for the `shuffle` enclave is $O(1)$ per request.

## 4.4 Optimizations

We further propose optimizations to Melbourne shuffle algorithm such that the performance can be improved by a constant factor. Both these optimizations are possible by exploiting the design structure of the algorithm.

**Pipelining.** We observe that in the existing algorithm (shown in Algorithm 1) the three phases execute sequentially (see Figure 4). Once the `dist_phase1` function generates the `temp1` array, it waits until the remaining phases complete. On the other hand, the `dist_phase2` and the `cleanup_phase` functions have to wait for the previous phase to complete before starting their execution. To eliminate this waiting time, we separate the execution of these phases into different enclaves and execute them in a pipeline. Thus, instead of waiting for the entire shuffle algorithm to complete, `dist_phase1` enclave generates a new `temp` array in every round to be used as input by the `dist_phase2` enclave. Eventually, each phase enclave outputs an array in every round to be used by the next phase enclave, thereby pipelining the entire execution. Note that this optimization is possible because the input to the `dist_phase1` does not depend on any other phase. `dist_phase1` enclave can continue to use the initial uploaded data as input and generate different `temp` arrays based on a new permutation value selected randomly in each round. This allows us to continuously execute each of the phases in its own enclave without becoming a bottleneck on any other phase. Thus, the overall latency is reduced by a factor of 3. This optimization increases the external storage requirement by $2N$ to store the additional temp array.

**Parallel Rounds using Multiple Enclaves.** Another optimization is to instantiate multiple (possibly $O(\sqrt{N})$) enclaves and execute each of the $O(\sqrt{N})$ rounds in parallel in these enclaves. With this optimization, the latency for shuffling $N$ data blocks reduces from $O(\sqrt{N})$ to $O(1)$. This observation is also discussed by Ohrimenko et al. [33]. However, the main drawback in implementing this optimization is the blow-up in the combined private storage. As each of $O(\sqrt{N})$ enclaves requires private memory of size $O(\sqrt{N})$, the combined private memory is linear in the total data size $O(N)$. Such a huge requirement of private storage may not be feasible even on very high-end servers. In our work, to use this optimization without requiring linear private storage, we propose using only a constant number of enclaves, thereby improving the performance by a constant factor.

## 5 Security Analysis

The observable access patterns in PRO-ORAM include accesses made both from access and shuffle enclave. We first show that the shuffle enclave executes an oblivious algorithm.

**Lemma 5.1.** *Given N data blocks, Melbourne Shuffle is an oblivious algorithm and generates a permuted array with very high probability* $(1-negl(N))$ *in* $O(\sqrt{N})$ *steps, each exchanging a message size of* $O(\sqrt{N})$ *between a private memory of* $O(\sqrt{N})$ *and untrusted storage of size* $O(N)$.

This Lemma directly follows from Theorem 5.1 and 5.6 in [33]. In PRO-ORAM, the shuffle enclave executes the Melbourne Shuffle algorithm using $O(\sqrt{N})$ memory within an enclave. Thus, from Lemma 5.1, we get the corollary below,

**Corollary 5.1.** *The* shuffle *enclave generates a* permuted *array of* $O(N)$ *data blocks in* $O(\sqrt{N})$ *steps and the access patterns are oblivious to the server.*

From Corollary 5.1, the access patterns of the shuffle enclave are oblivious and the output is indistinguishable from a pseudo-random permutation (PRP) [33].

Further, the communication between access and shuffle enclave happens over a secure channel. This preserves the confidentiality of the permutation and encryption keys that shuffle enclave sends to the access enclave at the end of each round. Thus, no information is leaked due to the interaction between these enclaves in PRO-ORAM. Now, to prove that PRO-ORAM guarantees obliviousness for read access patterns, we first show that a request to the access enclave is indistinguishable from random for an adaptive adversary.

Let $\mathscr{E} = (Gen, Enc, Dec)$ be a IND-CPA secure encryption scheme where *Gen* generates a key which is used by the *Enc* and *Dec* algorithms to perform encryption and decryption respectively. Let $\lambda$ be the security parameter used in $\mathscr{E}$. $\text{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}$ refers to the instantiation of the experiment with PRO-ORAM, $\mathscr{E}$ algorithms and adaptive adversary $\mathscr{A}_{adt}$. This experiment captures our security definition for read-only obliviousness. The experiment consists of:

- $\mathscr{A}_{adt}$ creates request $r = (read, d_i)$ and sends it to a challenger $\mathscr{C}$.
- The challenger selects $b \xleftarrow{\$} \{1,0\}$.
- If $\mathtt{b} = 1$, then $\mathscr{C}$ outputs the address access patterns to fetch $d_i$ i.e., $A(d_1) \leftarrow$ access $(d_i)$ and encrypted output $O'_1 \leftarrow d'_i$
- If $\mathtt{b} = 0$, then $\mathscr{C}$ outputs a random address access pattern i.e., $A(d_0) \xleftarrow{\$} \{1, \cdots, N+\sqrt{N}\}$ and $O'_0 \xleftarrow{\$} \{0,1\}^{\lambda}$
- Adversary $\mathscr{A}_{adt}$ has access to an oracle $\mathscr{O}^{\texttt{PRO-ORAM}}$ that issues $q$ queries of type $(read, d)$ both before and after executing the challenge query $r$. The oracle outputs address access patterns to fetch $d$ i.e., $A(d) \leftarrow$ access $(d)$ .
- $\mathscr{A}_{adt}$ outputs $\mathtt{b}' \in \{1,0\}$.
- The output of the experiment is 1 if $\mathtt{b} = \mathtt{b}'$ otherwise 0. The adversary $\mathscr{A}_{adt}$ wins if $\text{Exp}^{\texttt{PO}}_{\mathscr{E}}(\lambda, \mathtt{b}') = 1$ .

Based on the experiment and its output, we define read-only obliviousness as follows:

**Definition 5.1.** An algorithm satisfies read-only obliviousness *iff* for all PPT adversaries $\mathscr{A}$, there exists a negligible function negl such that:

$$Pr[\text{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,1) = 1] - Pr[\text{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,0) = 1] \leq \texttt{negl} \quad (1)$$

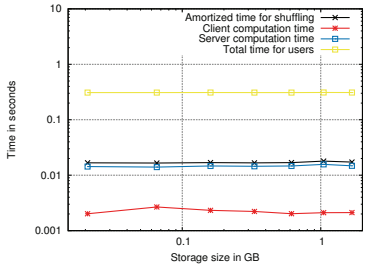
**Theorem 5.1.** *If* shuffle *enclave executes an oblivious algorithm and* $\mathscr{E}$ *is a CPA-secure symmetric encryption then* PRO-ORAM *guarantees read-only obliviousness as in Def. 5.1.*

*Proof.* We present the proof in Appendix A

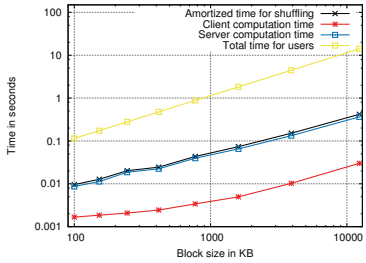
## 6 Implementation and Evaluation

**Implementation.** We have implemented our proposed PRO-ORAM algorithm in C/C++ using Intel SGX Linux SDK v1.8 [9]. For implementing symmetric and public key encryption schemes, we use AES with 128-bit keys and Elgamal cryptosystem with 2048 bit key size respectively from the OpenSSL library [39]. We use SHA256 as our hash function. We implement the read logic of square-root ORAM and the parallelized shuffle algorithm as explained in Section 4.2. We use multi-threading with SGX enclaves to implement our parallel execution approach for each step. The prototype contains total 4184 lines of code measured using CLOC tool [3].

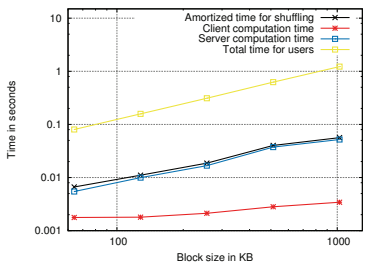
**Experimental Setup & Methodology.** To evaluate PRO-ORAM, we use SGX enclaves using the Intel SGX simulator and perform experiments on a server running Ubuntu 16.04 with Intel(R) Xeon(R) CPU E5-2640 v4 processors running at 2.4 GHz (40 cores) and 128 GB of RAM. As PRO-ORAM's design uses $\sqrt{N}$ threads, our experimental setup of 40 cores can execute a total of 80 threads using Intel's support for Hyper-Threading, thereby handling requests with block-size of 256 KB for around 1 GB of data. Operating with data of this size is not possible with SGX in hardware mode available on laptops due to their limited processing capacity (8 cores). However, for real cloud deployments, the cost of a deca-core server is about a thousand dollars [10]; so, the one-time cost of buying 40 cores worth of computation per GB seems reasonable. To measure our results for gigabyte sized data, we chose to run 40 cores (80 threads) each with an SGX simulator.




(a) Execution time is constant for fixed $B = 256KB$.

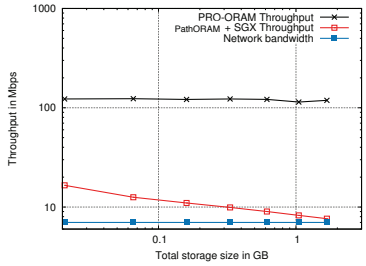



(b) Execution increases with $B$ for $N.B = 1GB$.

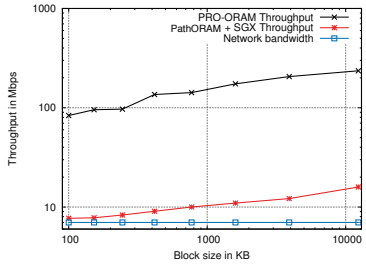



(c) Execution time increases with $B$ where $N = 4096$.

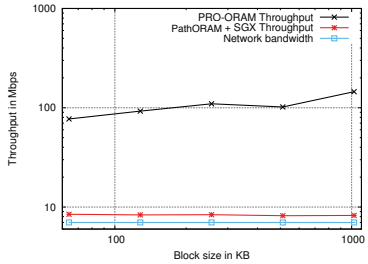
Figure 5: Execution time for client, server, shuffle and total latency per access for a fixed block size ($B$), fixed total storage ($N.B$) and a fixed no. of blocks ($N$)




(a) Throughput for varying $N.B$ where $B = 256KB$.




(b) Throughput for varying $B$ where $N.B = 1GB$.




(c) Throughput for varying $B$ where $N = 4096$.

Figure 6: Throughput of PRO-ORAM in Mbps for fixed block size ($B$), fixed total storage ($N.B$) and fixed number of blocks ($N$)

As a baseline for comparisons of communication and network latencies, we take the bandwidth link of 7 Mbps as a representative, which is the global average based on a recent report from Akamai [7]. We perform our evaluation on varying data sizes such that the total data ranges from 20 MB to 2 GB with block sizes ($B$) varying from 4 KB to 10 MB. In our experiments for parallelized shuffle, as shown in Algorithm 1, we set temporary buffers as $2\sqrt{N}$ data blocks to ensure security guarantees. To make full use of computation power, we utilize all 40 cores for performing multi-threading for each distribution phase and cleanup phase. All results are averaged over 10 runs, reported on log-scale plots. We perform our evaluation with the following goals:

- To validate our theoretical claim of constant communication and computation latencies.
- To confirm that execution time per access in PRO-ORAM is dependent *only* on the block size.
- To show that the final bottleneck is the network latency, rather than computational latency.
- To show the effective throughput of PRO-ORAM for different blocks of practical data sizes.

## 6.1 Results: Latency

To measure the performance, we calculate the execution time (latency) at the user, server (i.e., `access` enclave) and the amortized shuffling time of the `shuffle` enclave for each request. We point out that the client computational latency, the amortized shuffle time, and the network latency are the three factors that add up to the overall latency.

**Impact on Latency with Increasing Storage.** We measure the execution time to access a block of fixed size $B = 256KB$, while increasing the total storage size from 20 MB to $2GB$. The measurements are reported in Figure 5a. The dominant cost, as expected, is from the server computation. The `access` and `shuffle` enclave each incur a constant execution time of around 0.016 seconds per access, irrespective of the data sizes. The client computation time is constant at 0.002 seconds as the user only decrypts a constant-size encrypted block. Overall, these results confirm our theoretical claims of constant latency per request, and that the latency for large data size (in GBs) is practical (under 1 sec for 256KB blocks).

**Computational vs. network bottleneck**. An important finding from Figure 5a is that the latency per access observed by the user is a constant at 0.3 seconds, within experimental error, irrespective of the total data size. Even though the server computation cost is high, the final latency has primary

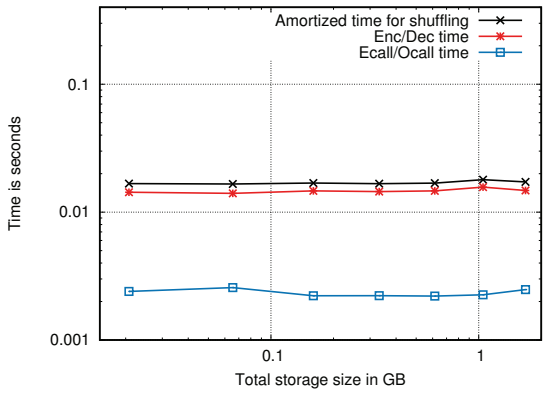



Figure 7: Overhead breakdown for shuffle step for fixed blocksize $B = 256$

bottleneck as the network, not PRO-ORAM's computation. In Figure 5a, the latency of shuffle per requested block is *lesser* than the network latency of sending a block from the server to the client on a 7Mbps link. This finding suggests that even for 256 KB block sizes, the network latency dominates the overall latency observed by the user, and is likely to be the bottleneck in an end application (e.g. streaming media) rather than the cost of all the operations in PRO-ORAM, including shuffling. This result suggests that PRO-ORAM is optimized enough to compete with network latency, making it practical to use in real applications.

**Latency increase with block size**. We perform three sets of experiments keeping (a) block size constant ($B$), (b) total storage size constant ($N.B$), and (c) number of blocks constant ($N$), while varying the remaining two parameters respectively in each experiment. The results in Figure 5b and 5c show evidence that the computational latencies of server and client-side cost in PRO-ORAM depend primarily on the block size parameter, and is unaffected by the number of blocks or size of data. This is mainly because the cost of encryption and decryption per block increases these latencies.

## 6.2 Results: Throughput

We calculate throughput as the number of bits that PRO-ORAM can serve per second. PRO-ORAM can serve maximum $\sqrt{N}$ blocks in the time the `shuffle` enclave completes permutation of $N$ data blocks. Thus, to calculate throughput we use the following formula, $\texttt{Throughput} = \frac{\sqrt{N}.B}{\texttt{total\_shuffling\_time}}$.

**Throughput increase with block size.** We find that throughput of PRO-ORAM increases with block size, ranging from 83 Mbps (for 100KB block size) to 235 Mbps (for 10MB block size), as shown in Figure 6b. Our experiments show that for data objects of the size larger than few hundred KB, the throughput is almost 10x larger than the global average network bandwidth (7Mbps). Such data object sizes are common for media content (e.g photos, videos, music) and cache web page content [11]. Figure 6b and Figure 6c show the throughput measurements for increasing block sizes, keeping the total data size and the number of blocks fixed to 1 GB and 4096 respectively. We observe that the throughput increases with the blocksize. If we keep the block size fixed, the throughput is constant at almost 125 Mbps with the increase in the total data size, as seen in Figure 6a. Our evaluation shows that PRO-ORAM's throughput exceeds reference throughput of 7 Mbps, re-confirming that network latency is likely to dominate latencies than computational overheads of PRO-ORAM.

**Comparison to Tree-based ORAM.** We compare the throughput of PRO-ORAM with the access overhead of using the simplest and efficient PathORAM scheme with SGX [43]. The client side operations in the original PathORAM scheme are executed within SGX. The throughput for PathORAM+SGX scheme decreases and almost reaches the network latency limit (7 Mbps) with increase in the number of blocks for fixed blocksize of 256 KB. Thus, the server computation overhead of $O(\log N)$ per access of PathORAM protocol becomes a bottleneck for reasonably large data sizes (e.g., 2 GB as shown in Figure 6a). Figure 6b shows that PathORAM's throughput increases from 7 to 15 Mbps with a decrease in blocks.

## 6.3 Performance Breakdown

To understand the breakdown of the source of latency for the shuffle step, we calculate the time to perform the cryptographic operations and ECALLs/OCALLs to copy data in and out of memory. Such a breakdown allows us to better understand the time consuming operations in our system. We fix the block size to $B = 256$ KB and vary the total data size. Figure 7 shows the amortized shuffling time, time to perform encryption and decryption operations and the time to invoke ECALLs/OCALLs per access in PRO-ORAM. We observe that the dominant cost comes from the cryptographic operations 0.014 seconds out of the 0.016 seconds. Enclaves by design cannot directly invoke system calls to access untrusted memory. Each call to the outside enclave performed using OCALL. Similarly, a function within an enclave is invoked using an ECALL. Thus, invocation of ECALLs/OCALLs is necessary to perform multi-threading and for copying data in and out of memory. To fetch $\sqrt{N}$ data blocks in parallel for each access, we use asynchronous ECALLs/OCALLs in PRO-ORAM similar to that proposed in a recent work [14]. These operations require 0.002 seconds (average) for a block of size 256 KB.

# 7 Related Work

First, we discuss ORAM constructions that guarantee constant latency per access for write-only patterns. Next, we summarize related work with similarities in our threat model.

**Write-Only ORAMs.** Recently, it is shown that constant computation and communication latency can be achieved

for applications with restricted patterns. Blass et. al show that some applications require hiding only write-patterns and hence proposed Write-Only ORAM in the context of hidden volumes [18]. Their work achieves constant latencies per write access to the data untrusted storage. Roche et al. propose a stash-free version of this Write-Only ORAM [35]. Further, Flat ORAM improves over this solution using secure processors to perform efficient memory management [24]. ObliviSync uses the write-only ORAM idea to support sharing of files on a Dropbox-like storage server that support auto-sync mechanisms [15]. These works that guarantee constant overhead for hiding write-only access patterns inspire our work. PRO-ORAM focuses on applications that exhibit read-only patterns and achieves constant latencies for them.

**Improvements to square-root ORAM.** Although square-root ORAM is known to have very high i.e., $O(N\log^2 N)$ worst-case overhead, Goodrich et. al provide a construction that reduces the worst-case overhead to $O(\sqrt{N}\log^2 N)$. Instead of shuffling the entire memory at once taking $O(N\log^2 N)$ computation time, their solution de-amortizes the computation over $\sqrt{N}$ batches each taking $O(\sqrt{N}\log^2 N)$ time after every access step. This technique is similar to the distribution of shuffle steps in PRO-ORAM. However, our observations for the read-only setting allows us to execute the access and shuffle steps in parallel which is not possible in their solution. Ohrimenko et. al show that use of Melbourne Shuffle combined with square-root ORAM can reduce the worst-case computation time to $O(\sqrt{N})$ with the use of $O(\sqrt{N})$ private memory. In PRO-ORAM, we show that it is further possible to reduce the latency to a constant for applications with read-heavy access patterns. Further, Zahur et al. have shown that although square-root ORAM has asymptotically worse results than the best known schemes, it can be modified to achieve efficient performance in multi-party computation as compared to the general optimized algorithms [46]. In PRO-ORAM, we have a similar observation where square-root ORAM approach performs better in the read-only setting.

**Solutions using Trusted Proxy.** ObliviStore [41] is the first work that uses a trusted proxy to mediate asynchronous accesses to shared data blocks among multiple users, which was later improved by TaoStore [36]. A major differentiating point is that both ObliviStore [41] and TaoStore [36] assume mutually trusting users that do not collude with the server, thus operating in a weaker threat model than ours. The key contribution in these works is towards improving efficiency using a single ORAM over having separate ORAMs for each user. ObliviStore improves the efficiency of the SSS ORAM protocol [42] by making ORAM operations asynchronous and parallel. Similar to this work, their key idea is to avoid blocking access requests on shuffle operations, thereby matching the rate of access and shuffle operations using a trusted proxy. However, their underlying approach to achieve such parallelization largely differs from this work. Our observations in designing PRO-ORAM are novel with respect to a read-only data setting that allows us to reduce the computation latency to a constant whereas ObliviStore has $O(\log N)$ computation latency per (read/write) access. TaoStore [36] is a more recent work that improves over ObliviStore using a trusted proxy and Path-ORAM [43] as its building block. Similar to [41], this approach has $O(\log N)$ computation latency per access.

**Solutions using Trusted Hardware.** An alternate line of work has shown the use of trusted hardware or secure processors with the goal to improve performance, as opposed to our use to strengthen existing ORAM protocols in a stronger threat model. Shroud uses trusted hardware to guarantee private access to large-scale data in data center [28]. ObliviAd is another system that makes use of trusted hardware to obliviously access advertisements from the server [16]. However, both these solutions do not optimize for read access patterns.

## 8 Conclusion

In this work, we provide a constant communication and computation latency solution to hide read data access patterns in a large class of cloud applications. PRO-ORAM guarantees a practical performance of 0.3 seconds to access a block of 256 KB leveraging sufficient storage and compute units with trusted hardware available on today's cloud platform. Our work demonstrates that simple ORAM solutions are better suited to hide read data access patterns than complex algorithms that are optimized for arbitrary read/write accesses.

## Acknowledgments

We thank the anonymous reviewers for their feedback. This research is supported in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21).

## References

[1] Software Guard Extensions Programming Reference. software.intel.com/sites/default/files/329298-001.pdf, Sept 2013.

[2] Box. https://www.box.com/home, Accessed: 2017.

[3] Cloc. http://cloc.sourceforge.net/, Accessed: 2017.

[4] Dropbox. https://www.dropbox.com/, Accessed: 2017.

[5] Dropbox hacked. https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach, Accessed: 2017.

[6] Dropbox usage statistics. http://expandedramblings.com/index.php/dropbox-statistics/, Accessed: 2017.

[7] Global average connection speed increases 26 percent year over year, according to akamai's 'fourth quarter, 2016 state of the internet report'. https://www.akamai.com/us/en/about/news/press/2017-press/akamai-releases-fourth-quarter-2016-/state-of-the-internet-connectivity-report.jsp, Accessed: 2017.

[8] Google drive. https://drive.google.com/drive/, Accessed: 2017.

[9] Intel sgx linux sdk. https://github.com/01org/linux-sgx, Accessed: 2017.

[10] Intel xeon processor pricing. https://ark.intel.com/products/92984/Intel-Xeon-Processor-E5-2640-v4-25M-Cache-2_40-GHz, Accessed: 2017.

[11] Web content statistics. http://httparchive.org/trends.php, Accessed: 2017.

[12] World's biggest data breaches. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/Internet_and_cloud_services_-_statistics_on_the_use_by_individuals, Accessed: 2017.

[13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.

[15] Adam J Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S Roche. Oblivisync: Practical oblivious file backup and synchronization. In *NDSS*, 2017.

[16] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 257–271. IEEE, 2012.

[17] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.

[18] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[19] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[20] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[21] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *USENIX Security*, volume 15, pages 447–462, 2015.

[22] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[23] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

[24] Syed Kamran Haider and Marten van Dijk. Flat oram: A simplified write-only oblivious ram construction for secure processor architectures. *arXiv preprint*, 2016.

[25] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12, 2012.

[26] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952*, 2016.

[27] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[28] Jacob R Lorch, Bryan Parno, James W Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST*, volume 2013, pages 199–213, 2013.

[29] Sinisa Matetic, Kari Kostiainen, Aritra Dhar, David Sommer, Mansoor Ahmed, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution. https://eprint.iacr.org/2017/048.pdf.

[30] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos 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*, page 10. ACM, 2016.

[31] Ming-Wei-Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS 2017*.

[32] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication oram with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873. ACM, 2015.

[33] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*, pages 556–567. Springer, 2014.

[34] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious ram. In *USENIX Security Symposium*, pages 415–430, 2015.

[35] Daniel S Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. Deterministic, stash-free write-only oram. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 507–521. ACM, 2017.

[36] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 198–217. IEEE, 2016.

[37] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. 2011.

[38] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328. ACM, 2016.

[39] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26-March 1, 2017*, 2017.

[40] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[41] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.

[42] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. In *NDSS'12*, 2011.

[43] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[44] Shruti Tople and Prateek Saxena. On the trade-offs in oblivious execution techniques. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017.

[45] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[46] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root oram: efficient random access in multi-party computation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 218–234. IEEE, 2016.

## A Security Analysis

**Theorem A.1.** *If* `shuffle` *enclave executes an oblivious algorithm and* $\mathscr{E}$ *is a CPA-secure symmetric encryption scheme then* `PRO-ORAM` *guarantees read-only obliviousness as in Definition 5.1.*

*Proof.* From Lemma 5.1, the access pattern of `shuffle` enclave are data-oblivious. To prove the theorem, we have to show that access pattern from `access` enclave are indistinguishable to the adversary. We proceed with a succession of games as follows:

- $\texttt{Game}_0$ is exactly the same as $\texttt{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,1)$
- $\texttt{Game}_1$ replaces the $O'_1$ in $\texttt{Game}_0$ with a random string while other parameters are the same

- $\texttt{Game}_2$ is same as $\texttt{Game}_1$ except that $A(d_i)$ is selected using a pseudorandom permutation $\pi_s : \{0,1\}^{(N+\sqrt{N})} \rightarrow \{0,1\}^{(N+\sqrt{N})}$ where $s \leftarrow \{0,1\}^{\lambda}$ and not from the `access` enclave.
- $\texttt{Game}_3$ is same as $\texttt{Game}_2$ except that $A(d_i)$ is selected at random from the entire data array.

From above description, we have

$$Pr[\texttt{Game}_0 = 1] = Pr[\texttt{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,1) = 1], \quad (2)$$

For $\texttt{Game}_1$, a distinguisher $D_1$ reduces the security of $\mathscr{E}$ to IND-CPA security such that:

$$Pr[\texttt{Game}_0 = 1] - Pr[\texttt{Game}_1 = 1] \leq Adv^{IND-CPA}_{D_1,\mathscr{E}}(\lambda), \quad (3)$$

For $\texttt{Game}_2$, according to Corollary 5.1, the advantage of a distinguisher $D_2$ is such that:

$$Pr[\texttt{Game}_1 = 1] - Pr[\texttt{Game}_2 = 1] \leq Adv^{Corollary\ 5.1}_{D_2,\texttt{shuffle}}, \quad (4)$$

This is because the `access` enclave uses the output of `shuffle` enclave to fetch the data for each request. The `access` enclave runs the square-root ORAM algorithm which selects a random address in each request. Hence, the advantage of the distinguisher $D_2$ depends on the correctness of the permuted output array from `shuffle` enclave.

For $\texttt{Game}_3$, a distinguisher $D_3$ reduces the security of $\pi$ to PRP security such that:

$$Pr[\texttt{Game}_2 = 1] - Pr[\texttt{Game}_3 = 1] \leq Adv^{PRP}_{D_3,\pi}(\lambda), \quad (5)$$

Also, we have,

$$Pr[\texttt{Game}_3 = 1] = Pr[\texttt{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,0) = 1], \quad (6)$$

From 2, 3, 4, 5, 6 we get:

$$Pr[\texttt{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,1) = 1] - Pr[\texttt{Exp}^{\texttt{PRO-ORAM}}_{\mathscr{A}_{adt},\mathscr{E}}(\lambda,0) = 1] \leq Adv^{IND-CPA}_{D_1,\mathscr{E}}(\lambda) + Adv^{Corollary\ 5.1}_{D_2,\texttt{shuffle}} + Adv^{PRP}_{D_3,\pi}(\lambda) \quad (7)$$

The $Adv^{Corollary\ 5.1}_{D_2,\texttt{shuffle}}$ cannot be greater than `negl` as it would break the security of the underlying Melbourne Shuffle algorithm stated in Lemma 5.1. With this, we prove that the advantage of an adaptive adversary in distinguishing the access patterns induced by `PRO-ORAM` from random is negligible. Therefore, `PRO-ORAM` guarantees read-only obliviousness.