MOSE: Practical Multi-User Oblivious Storage via Secure Enclaves

Thang Hoang
CSE, University of South Florida

Rouzbeh Behnia
CSE, University of South Florida

Yeongjin Jang
EECS, Oregon State University

Attila A. Yavuz
CSE, University of South Florida

ABSTRACT

Multi-user oblivious storage allows users to access their shared data on the cloud while retaining access pattern obliviousness and data confidentiality simultaneously. Most secure and efficient oblivious storage systems focus on the utilization of the maximum network bandwidth in serving concurrent accesses via a trusted proxy. However, since the proxy executes a standard ORAM protocol over the network, the performance is capped by the network bandwidth and latency. Moreover, some important features such as access control and security against active adversaries have not been thoroughly explored in such proxy settings.

In this paper, we propose MOSE, a multi-user oblivious storage system that is efficient and enjoys from some desirable security properties. Our main idea is to harness a secure enclave, namely Intel SGX, residing on the untrusted storage server to execute proxy logic, thereby, minimizing the network bottleneck of proxy-based designs. In this regard, we address various technical design challenges such as memory constraints, side-channel attacks and scalability issues when enabling proxy logic in the secure enclave. We present a formal security model and analysis for secure enclave multi-user ORAM with access control. We optimize MOSE to boost its throughput in serving concurrent requests. We implemented MOSE and evaluated its performance on commodity hardware. Our evaluation confirmed the efficiency of MOSE, where it achieves approximately two orders of magnitudes higher throughput than the state-of-the-art proxy-based design, and also, its performance is scalable proportional to the available system resources.

CCS CONCEPTS

• **Security and privacy** → *Privacy-preserving protocols.*

KEYWORDS

secure enclaves; multi-user ORAM; oblivious storage

ACM Reference Format:
Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A. Yavuz. 2020. MOSE: Practical Multi-User Oblivious Storage via Secure Enclaves. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20), March 16–18, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3374664.3375749

CODASPY '20, March 16–18, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7107-0/20/03…$15.00
https://doi.org/10.1145/3374664.3375749

1 INTRODUCTION

Data outsourcing is a promising approach in the cloud era for efficiently providing a large amount of storage to many remote clients. However, storing data on untrusted servers raises security/privacy concerns. Applying encryption to data may provide data confidentiality, but it is not sufficient to address such concerns. In particular, leaking the access patterns may put data privacy at risk. Once clients access their data on the cloud, they leak access patterns to the cloud provider as well as the attackers on the network. As a consequence, recently, a number of attacks exploiting such access pattern leakage have been demonstrated, especially in the encrypted search domain [7, 20, 27, 33, 56].

Standard ORAM in the single-user setting. Oblivious Random Access Machine (ORAM) [18] is a promising technique for data outsourcing because ORAM can hide access patterns, but to use it, one should overcome its significant computation and/or network bandwidth overhead. Despite the recent algorithmic improvements in ORAM [22, 38, 42, 46, 47, 51], there exists a limit in network bandwidth usage: logarithmic communication lower bound. In client-server settings, such communication is performed over the network with limited bandwidth and high latency [4, 22]. Several ORAM schemes with $O(1)$ bandwidth (e.g., [2, 12, 15]) were proposed; however, they incur high server computation overhead.

Multi-user ORAM. ORAM could be extended to support multi-user settings, but doing so brings up new challenges such as network bandwidth overhead, complexity of handling concurrency, asynchronicity, etc. Once the single-user ORAM is adopted into the multi-user setting, the impact of network delay drastically increases since user requests must be processed in a sequential order over the network. Alternatives were proposed to enable some security and efficiency properties such as access pattern obliviousness, access control and concurrency. Unfortunately, existing techniques do not fully satisfy all the performance and security requirements. One research direction focuses on proposing new ORAM with access control (e.g., [30, 31]), or parallelization (e.g., [6, 8, 9, 35]). Although these schemes do not rely on an additional trusted party (e.g., a proxy), they incur either high network communication (e.g., $O(\log^2 N)$ where N is the number of data blocks) (e.g., [6, 8, 9]) or server computation overhead [31] (i.e., $O(N)$). Maffei et al. in [31] showed that there exists a computation lower bound of $\Omega(N)$ to achieve both the access pattern obliviousness and access control at the same time against an active adversary.

Another research direction leveraged a trusted proxy to handle multi-user concurrent accesses and access control (suggested in [31]) by using a standard ORAM protocol [4, 40, 44, 45, 54]. Unfortunately, fully asynchronous proxy designs (e.g., [4, 45]) were shown insecure against timing attacks [40] exploiting access pattern leakages in processing concurrent requests. To fix this vulnerability,

Sahin et al. proposed TaoStore [40], the most efficient and secure proxy design to-date, which allows ORAM protocols to be securely executed in parallel over the network. Despite their merits, all proxy designs execute standard ORAM protocols over the network so that their performance is capped by the limitations of the network bandwidth and latency, and fail to scale beyond that limit (e.g., TaoStore supports up to *ten* users for a 1Gbps link). Since all proxy designs are originally proposed to exploit the maximum network throughput for concurrent accesses, they do not focus on access control enforcement or handling an active adversary. Maffei et al. in [31] pointed out that proxy-based designs can offer access control enforcement with the incorporation of an access control data structure and authentication at the proxy. A detailed exploration of this enhancement in practical settings is a worthy investigation.

Our Approach. After observing the drawbacks of the previous approaches, we aim to achieve three major goals for a secure and efficient data outsourcing: *1) Low network overhead*; *2) Access control and security assurances against active adversary*; and *3) Secure parallelization at the server side for fast concurrent access.*

To this end, we design our system, MOSE, which employs secure enclave to drastically reduce network overhead, support security functionalities against active adversaries, and exploit parallelism in the server at once. In particular, we first develop the trusted logics (similar to the trusted proxy) on an untrusted server (i.e., not on the network) by adopting a commodity secure enclave (Intel SGX) as a secure isolation mechanism. This design is inspired by recent systems and frameworks assisted by Intel SGX (e.g., [1, 13, 14, 16, 23, 36, 41]), which employ a commodity secure hardware to enhance the efficiency of the underlying cryptographic operations. We are motivated by these approaches to overcome network bandwidth limitations of the previous proxy-assisted ORAM systems.

Contributions. MOSE offers the following desirable properties.

• *Minimal network bandwidth overhead:* MOSE removes network-related problems in existing proxy-assisted oblivious storage systems by putting the trusted logic on an enclave. In this setting, the enclave communicates with the storage server via a local memory bus, which is several orders of magnitude faster than the network communication in terms of both bandwidth and latency. This design does not incur network bandwidth overhead because MOSE will transfer only the encrypted, bulk data over the network.

• *Access control enforcement against active adversary:* MOSE, running in a secure enclave, securely enforces access control policies and hides the access patterns against an active adversary corrupting the users and/or all components of the server except for the CPU. To this end, we define a security notion for enclave-assisted multi-user ORAM with access control in the presence of an active adversary. We show that MOSE achieves the security according to this model (see §6).

• *Scalable to multi-user concurrent access:* MOSE achieves a scalable performance by utilizing a parallel optimization, which achieves maximum concurrency at the server while preserving the ORAM security. In particular, MOSE is scalable proportional to the available system resources, e.g., the number of CPU cores in the server. These properties allow MOSE to achieve a high throughput in serving a large number of concurrent requests from multiple users.

• *Implementation and experimental evaluations:* We implemented MOSE and evaluated its performance on a commodity desktop that supports Intel SGX, and the experimental result indicated that our system is efficient and practical (see §7). For example, with 96 GB database, MOSE can process 374 concurrent requests while achieving less than one second delay to all requests, and it can process 788 concurrent requests while achieving, on average, one second delay per request. MOSE is two orders of magnitudes faster than the state-of-the-art [40] and thus more suitable in serving multi-users requests. Specifically, MOSE can support more than 300 concurrent users with a reasonable delay for accessing 96 GB database, compared with 10 users on 13 GB database in TaoStore with 1 Gbps network bandwidth. To achieve these, MOSE incurs 10 GB of additional memory usage and doubles the storage overhead.

Based on these results, we believe that MOSE fills an important practical gap between security and performance, and facilitates secure data outsourcing in practice.

2 DESIGN DECISIONS AND CHALLENGES

We discuss design requirements and challenges in achieving an efficient and secure multi-user oblivious storage using secure enclaves.

2.1 Requirements

A secure data outsourcing method is expected to meet the following requirements to ensure practicality and security.

Performance: To allow multiple users to access the shared database in a timely manner, the system should achieve the following performance requirements:

(1) **Efficiency**: The system must not incur significant computation delay to achieve a reasonably fast response time (i.e., latency) and does not incur huge overhead in network bandwidth.

(2) **Scalability**: To serve arbitrarily many concurrent user requests, the system should be able to scale its performance proportional to the available system resources.

Security: In addition to the data confidentiality provided by encryption, guaranteeing the following three security properties is required for securing outsourced database in the multi-user setting:

(1) **Obliviousness**: The system should not leak any meaningful information from users' access patterns.

(2) **Access control**: The system should be able to check whether a user has access to a specific entry in the database while retaining the obliviousness of users' access patterns.

(3) **Resiliency against active adversaries**: The system should ensure that both data and access patterns of honest users are not leaked to the malicious server, active attackers in the network or compromised users, even if they collude with each other.

2.2 Why Secure Enclaves?

We decided to adopt secure enclaves in MOSE's design based on the following reasons. Both previous directions for oblivious access in the multi-user setting (i.e., non-proxy ORAM and proxy-based ORAM) suffer from distinct limitations. Without a proxy, a purely cryptographic ORAM construction requires a $\Omega(N)$ computation to enable access pattern obliviousness and access control at the same time against an active adversary [31]. In contrast, any proxy-based oblivious storage can enable access control by maintaining an access control data structure at the proxy. However, their network bandwidth overhead is inevitably costly because the ORAM

protocol is executed over the network. Therefore, even for the state-of-the-art implementation [40], only *ten* concurrent users can be supported in 1 Gbps network, capped by the bandwidth limit, and this renders TaoStore impractical to use in real-world scenarios (e.g., running a web server).

Our observation is that the designs without proxy incur extreme computation overhead that might be impractical for large databases, and the performance of the designs with proxy is restricted by the network bandwidth. Focusing on the latter case, which is more feasible than the former, we come up with a following research question: *Can we remove the restraining network layer in proxy-based designs between the trusted proxy and the storage server?*

Inspired by recent Intel SGX-assisted frameworks [1, 14, 16, 23, 36, 41], we aim to utilize the secure enclave to run our trusted proxy to overcome the network bandwidth limitation. We target storage-optimized infrastructures, in which our construction utilizes high-speed local links (e.g., Infiniband/PCIe) for oblivious access to the storage units while using remote link (i.e., Internet) only to transfer the payload. These infrastructures are expected to be widely available on commodity cloud services with enhanced security features (e.g., MS Azure Confidential Computing). Secure enclave, specifically Intel SGX, is a hardware-based isolation mechanism that provides a trusted execution environment and, therefore, it can work as an isolation mechanism of a trusted proxy when running on an untrusted server. The benefit of the enclave is that the communication channel between the proxy and the server is now a *local memory bus*, which is orders of magnitude faster than network links (273 Gbps *vs.* 1 Gbps). The enclave can execute trusted logics such as access control, which also meets our security demand.

2.3 Challenges

Putting the trusted proxy in an enclave will remove the performance bottleneck in the proxy-based ORAM and enable access control functionality. However, designing the system that meets all the aforementioned requirements for data outsourcing with secure enclave is a challenging task. We highlight some of these major design challenges in the following.

System-level restrictions. Security and isolation guarantees provided by the secure enclave come with a performance cost, and our design should be able to handle such performance issues efficiently. First, because the enclave provided by Intel SGX can only protect user-level processes, and the operating system is untrusted in the Intel SGX security model, invoking any system calls that move data outside the security domain requires encryption. Particularly in MOSE, each I/O access that reads/writes the untrusted memory as well as network communication requires encryption, thereby the system must handle this accurately and efficiently. Second, Intel SGX is performant when an enclave manages its active memory usage < 95 MB due to the hardware limit. A study [11] shows that a program could suffer more than 200% slowdown, if its enclave uses memory beyond that boundary. Therefore, designs that include a large amount of data (e.g., storing 8 GB data in trusted memory) cannot meet the performance requirement.

Side-channel attacks against secure enclaves. The design must protect the trusted logic executed by an enclave from side-channel attacks because Intel SGX does not guarantee the security against such attacks. In particular, Intel SGX does not provide any protection for memory access pattern even for the trusted memory. In this regard, researchers have demonstrated that attackers could learn about the data that an enclave processes via cache [10, 19, 26, 32, 48, 50, 53] or page-fault [21, 49, 55] side-channel. Although the oblivious data access requirement of MOSE can protect side-channel attacks when reading data from storage, the enclave cannot automatically protect the security critical logics. For instance, when an ORAM controller reads a block from a path, attacker could learn the position of a block from the enclave's execution and memory access pattern. Therefore, the design must ensure its resiliency against side-channel attacks, as long as the attack is not caused by CPU misimplementation [10, 48, 50, 53].

Achieving scalable performance. To meet the scalability requirement, the design should be able to scale its performance to the available system resources. While putting the trusted proxy in the enclave unleashes the performance bottleneck caused by network bandwidth limit, it does not guarantee that the proxy can utilize system's full bandwidth. Since the memory bus is fast, adopting the enclave will change the performance limiting factor to other factors, such as storage I/O access, computation speed for cryptographic operations, etc. Hence, the design must analyze and identify performance hurdles in this architecture and tailor itself to be scalable. Additionally, scalable design should also consider that the design is free from concurrency attacks to ORAM [40].

3 MOSE HIGH-LEVEL ARCHITECTURE

3.1 Overview

At a high level, MOSE is an outsourced oblivious storage in an untrusted cloud server with a trusted proxy that handles user's requests. MOSE runs its trusted logic such as user authentication, en/decryption, and the ORAM controller logic in the enclave. Unlike previous approaches that put the trusted proxy in a physically separated server on the network, MOSE utilizes hardware-based isolation provided by Intel SGX and can guarantee a comparable grade of security to such approaches. To make an access request to MOSE, a user must be authenticated by the enclave logic and also pass the access control requirements. After that, the enclave will perform oblivious access to the encrypted database stored in untrusted storage. The trusted logic will decrypt the content, and finally, the result will be returned to the user. Note that the communication between the user and the enclave is encrypted by the secret key shared by the remote attestation protocol provided by Intel SGX.

On the other hand, the storage is untrusted, thereby the database and the access control structure must be encrypted. This proxy construction ensures *three* security requirements for data outsourcing. First, the ORAM access from trusted proxy can guarantee the access pattern obliviousness if our secure enclave does not leak any critical data via side-channel analysis. Second, to guarantee security, MOSE blocks control-flow side-channel via secure (non-branch) comparison and assignment logic and blocks cache side-channel by accessing the entire data for critical ORAM components such as requested paths and the stash. Third, for access control and resiliency, MOSE implements access control logic in the trusted enclave and uses ORAM to obliviously check the permission, so even when a malicious user colludes with the server, they cannot infer any information about the honest user(s).

Regarding the performance, MOSE achieves the two following design requirements. First, by having a trusted proxy in a secure

enclave, MOSE eliminates efficiency issues related to network bandwidth/latency applied between the proxy and the storage server. Instead of communicating via a network link, the trusted proxy of MOSE uses memory bus, which is two orders of magnitude faster (273 Gbps *vs.* 1 Gbps) in bandwidth and much faster (350 ns *vs.* $\approx$ 50 ms) in latency. This design unleashes the bottleneck in serving concurrent requests as in TaoStore [40], which can only support *ten* users for achieving their optimal performance.

Second, MOSE offers scalable performance for supporting multi-user access by speeding up a single ORAM access via parallelization while processing the entire user access requests sequentially. Unlike previous works aiming at parallelizing multiple ORAM access in a concurrent manner, this construction makes MOSE free from the asynchronicity attack [40]. In particular, MOSE applies parallelization to a single ORAM access request regarding encryption and decryption computations, and I/O access. MOSE splits each block in ORAM structures into m chunks, where m is the number of parallel threads. When processing an access request, each thread processes the reading of each chunk from the storage as well as cryptographic computation in parallel. Thanks to this parallel construction, MOSE's performance improves in proportion to the number of parallel threads, and therefore, it is scalable.

3.2 System Model

Our system is comprised of a data owner, k users, and an untrusted storage server $\mathcal{S}$ equipped with Intel SGX. The data owner owns a database (DB) containing N blocks and grants permissions such as read (R), read&write (RW) for k users to access N blocks via an access control data structure (AC). Only the data owner can update the AC entries. The data owner encrypts DB and AC forming EDB and EAC, respectively, both of which are stored in the untrusted memory region of $\mathcal{S}$ such as solid-state drive (SSD). To access an entry in DB/AC, the users/data owner interacts with an enclave created by Intel SGX, which acts as a trusted proxy to execute ORAM operations with $\mathcal{S}$.

For the sake of simplicity, we say accessing $\mathcal{S}$ to imply accessing the untrusted memory region in $\mathcal{S}$. We consider the enclave as the ORAM client (OClient) in our models because it is the only entity that executes the ORAM protocol with $\mathcal{S}$.

Inspired by [31, 40], we present the definition of multi-user ORAM with trusted proxy in Definition 3.1. Our model differs from the ones in [31] and [40] in the sense that the former does not use proxy while the latter does not consider the access control enforcement. We denote the execution of protocol A by OClient with the server $\mathcal{S}$ as $(\vec{o}_C; \vec{o}_S) \leftarrow A(\vec{i}_C; \vec{i}_S)$, where the input/output vectors of two parties are separated by a semicolon (;).

Definition 3.1 (Enclave-assisted multi-user ORAM with access control). A multi-user ORAM with access control is comprised of the following (interactive) Probabilistic Polynomial Time (PPT) algorithms:

- $(k_o, \mathsf{EDB}, \mathsf{EAC}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{DB}, k, N)$: It takes as input a security parameter λ, and a database DB containing N data blocks. It initializes a data structure AC managing the access policy of k users for N blocks. It returns EDB and EAC as the encrypted form of DB and AC, respectively.
- $(p; \perp) \leftarrow \mathsf{ReadAC}(\mathsf{uid}, \mathsf{bid}; \mathsf{EAC})$: It takes a user ID uid and a block ID bid from OClient and EAC from $\mathcal{S}$ as input. It reads the permission on EAC for entries bid and uid as $p \leftarrow \mathsf{EAC}(\mathsf{uid}, \mathsf{bid})$. It outputs a permission $p \in \{\mathsf{R}, \mathsf{RW}, \perp\}$ to OClient and $\perp$ to $\mathcal{S}$.
- $(p; \mathsf{EAC}') \leftarrow \mathsf{WriteAC}(\mathsf{uid}, \mathsf{bid}, p; \mathsf{EAC})$: It takes (uid, bid, p) from OClient, where $p \in \{\mathsf{R}, \mathsf{W}, \mathsf{RW}, \perp\}$ and EAC from $\mathcal{S}$ as input. It updates as $\mathsf{EAC}(\mathsf{uid}, \mathsf{bid}) \leftarrow p$. It outputs p to OClient, and the updated EAC$'$ to $\mathcal{S}$.
- $(\mathsf{data}; \perp) \leftarrow \mathsf{ReadDB}(\mathsf{uid}, \mathsf{bid}; \mathsf{EAC}, \mathsf{EDB})$: It takes (uid, bid) from OClient and (EAC, EDB) from $\mathcal{S}$ as input. It executes $(p; \perp) \leftarrow \mathsf{ReadAC}(\mathsf{uid}, \mathsf{bid}; \mathsf{EAC})$. If $p \notin \{\mathsf{R}, \mathsf{RW}\}$, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[i]$, where i is a dummy block ID. Otherwise, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[\mathsf{bid}]$. Finally, it returns data to OClient and $\perp$ to $\mathcal{S}$.
- $(\mathsf{data}; \mathsf{EDB}') \leftarrow \mathsf{WriteDB}(\mathsf{uid}, \mathsf{bid}, \mathsf{data}^*; \mathsf{EAC}, \mathsf{EDB})$: It takes (uid, bid, data^*) from OClient, where data^* is the new data to be written, and (EDB, EAC) from $\mathcal{S}$ as input. It executes $(p, \perp) \leftarrow \mathsf{ReadAC}(\mathsf{uid}, \mathsf{bid}; \mathsf{EAC})$. If $p \neq \mathsf{RW}$, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[i]$, where i is an dummy block ID. Otherwise, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[\mathsf{bid}]$, and updates $\mathsf{EDB}[\mathsf{bid}] \leftarrow \mathsf{data}^*$. It returns data to OClient, and EDB$'$ to $\mathcal{S}$ indicating EDB is (possibly) updated.
- $\mathsf{data}' \leftarrow \mathsf{Response}(k_{\mathsf{uid}}, \mathsf{data})$: It encrypts data with key k_{uid}.

3.3 Threat Model

We build MOSE based on following assumptions as its threat model.
• *Trusted:* We trust the data owner, OClient and the ORAM controller logic, both of which run by the enclave. We trust the remote attestation protocol by Intel SGX, on which we rely on for verifying the integrity of the enclave and establishing a secure communication channel between the user and the enclave. We also trust the hardware key, generated by the enclave that seals the ORAM key.
• *Untrusted:* We assume that the server $\mathcal{S}$ is completely untrusted except the enclave. We assume attackers on the server can freely monitor access patterns of the users and the enclave such as network access, storage and memory access; however, they cannot compromise data confidentiality for such accesses. We do not trust any of server's logic that includes virtual machine monitor, operating system and drivers, software that manages storage, etc. This is a general assumption for a system that utilizes secure enclaves because Intel SGX isolates and applies encryption to the enclave's memory space via hardware mechanisms. Additionally, data users and the server can also be *active* adversaries, meaning that they may attempt to inject/tamper their input or even collude with each other to break the security of other honest users.

4 BUILDING BLOCKS

Notation. $x \xleftarrow{\$} \mathcal{S}$ denotes x is uniformly and randomly sampled from set S. We denote $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ as an IND-CPA-secure symmetric encryption, where $k \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$ generates a symmetric key k given a security parameter 1^λ; $c \leftarrow \mathcal{E}.\mathsf{Enc}_k(M)$ returns the ciphertext c of M encrypted with key k; $M \leftarrow \mathcal{E}.\mathsf{Dec}_k(c)$ returns the plaintext M of c, which is previously encrypted by k.

4.1 Intel SGX

We use Intel SGX as a secure enclave to securely execute the trusted logic on an untrusted server. In the following, we describe how we exploit the hardware-based isolation for secure execution and its remote attestation for establishing a secure communication channel between a client and the trusted logic. We also describe how we

securely use Intel SGX while providing defense against existing threats, such as untrusted system calls.

Isolation by secure enclave. Intel SGX provides an *Enclave*, which isolates its memory from untrusted system components other than CPU and also protects the integrity and confidentiality of its memory. Its hardware-based isolation does not allow any execution other than the program runs in the enclave from accessing enclave's data. We run MOSE's trusted logic for processing the access control and ORAM protocol in an enclave. Thus access to such logic is prevented from any untrusted part of the server machine.

Remote attestation and secure communication. We utilize the remote attestation protocol of Intel SGX to verify the integrity of our trusted logic and also securely exchange secret keys for protecting communication channels between the trusted logic and the user. With this, we ensure that clients do not directly communicate with the server in the case of a failed integrity check and therefore, reduce our attack surface for man-in-the-middle attack. In addition to the measurement, the protocol also securely shares Diffie-Hellman key parameters. We use this shared secret as a session key to protect the communication channel between the trusted logic and the user.

Use encryption for untrusted system calls. System calls including network and disk I/O are untrusted in Intel SGX. Following the guidelines provided by Intel and security community [11, 24, 25], we apply encryption on the data if the data moves across the security domains (i.e., encrypts data if they are being processed by system calls, decrypts data if they are being processed by an enclave). Therefore, security components of MOSE such as the position map and plaintext data of database are stored encrypted on the untrusted server (thus secure), and they will be decrypted by our trusted logic in an enclave only when they are being accessed by a user request.

4.2 Circuit-ORAM

ORAM allows the client to hide access patterns when accessing an encrypted database stored in the untrusted memory region [18]. Most efficient ORAM schemes for client-server applications [22, 38, 47, 51] follow the tree-paradigm proposed in [42]. Among these, we select Circuit-ORAM [51] for MOSE since it features the optimal circuit size and therefore, it is the most efficient to be incorporated with the secure enclave technique.

Definition 4.1 (Circuit-ORAM [51]). It comprises two PPT algorithms as follows.

- $\mathsf{EDB} \leftarrow \mathsf{Setup}(k, \mathsf{DB})$: Given a key k and a database DB with N data blocks as input, it first generates a binary tree $\mathbf{T}_0$ with N leaves, in which each data block is assigned to a random path and resided somewhere along the assigned path. It also generates a stash component (S) of size $O(|B|\log N)$, where $|B|$ is the block size, and a position map (pos) of size $O(N \log N)$, which stores the path information of each block. It then store pos in a series of small binary trees $(\mathbf{T}_1, \ldots, \mathbf{T}_{\log N})$ with recursive technique in [42]. Finally, it outputs an encrypted database EDB, which includes the encrypted version of all the aforementioned components as $\mathsf{EDB} \leftarrow \left(\{\mathcal{E}.\mathsf{Enc}_k(\mathbf{T}_i)\}_{i=0}^{\log N}, \mathcal{E}.\mathsf{Enc}_k(S)\right)$.
- $\mathsf{data} \leftarrow \mathsf{Access}(\mathsf{op}, \mathsf{bid}, \mathsf{data}'; \mathsf{EDB})$: The client inputs an access operation $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, a block ID bid and data data' to be updated (if $\mathsf{op} = \mathsf{write}$). The server inputs the encrypted database EDB. It outputs the read data of bid (data) to the client.

Each Circuit-ORAM access comprises two following phases.

- *Read:* The client gets the path of bid from pos. She then reads and decrypts all nodes along the path pid from the ORAM tree, and puts only bid into S. She removes bid info from the path and updates the path of bid with a new path selected uniformly at random.
- *Eviction:* The client selects an eviction path randomly as in [17]. The client prepares a list of blocks to be moved in the eviction path. The client picks one block from S that can be pushed to the deepest level of the tree, and then traverses from the root to leaf. In each level, the client drops the holding block and picks another block to be resided into a deeper level, until the leaf is reached.

Data Integrity. In MOSE, we employ authenticated encryption technique (i.e., AES-CTR for encryption and HMAC with SHA-256 for authentication) to achieve the integrity for each node in the Circuit-ORAM tree against malicious adversary.

5 DESIGN OF MOSE

5.1 System Initialization

Figure 1 presents the initialization workflow of MOSE. Given a database DB containing (upto) N block entries being shared among (upto) k users $(\mathsf{uid}_1, \ldots, \mathsf{uid}_k)$, the data owner executes the Gen algorithm to construct mandatory data structures outsourced to the cloud (Step 1). The algorithms first generates an ORAM symmetric key k_o and initializes an access control matrix AC (see §5.3 for our argument on why matrix structure is preferred) for each user uid_i with each database entry bid_j. For simplicity, we consider basic permission attributes including read and write. DB and AC are then packaged and encrypted into two separate recursive Circuit-ORAM structures called EDB and EAC, respectively. Figure 2 presents the Gen algorithm, which executes Circuit-ORAM setup algorithm in Theorem 4.1 to construct recursive Circuit-ORAM structures for the database and the access control structure. After the algorithm is executed, the data owner sends EAC and EDB as well as their encrypted position map (all in the form of Circuit-ORAM trees) to the server, all of which are stored in the untrusted memory region (e.g., SSD, at Step 2). The data owner performs a remote attestation of an enclave (provided by Intel SGX) running on the server to ensure if the enclave is intact and to exchange cryptographic key for establishing an encrypted communication channel between the data owner and the enclave. To this end, the data owner sends k_o to the enclave via the established channel (Step 3). To ensure security, the enclave always keeps k_o in the trusted memory, and stores it to the disk only after encrypting it with the enclave's hardware key (i.e., data sealing feature provided by Intel SGX).

5.2 Handling user request with access control

Figure 3 illustrates the workflow of MOSE in processing the user request with the access control check. Let bid be the ID of the block in EDB the user uid wants to access. The user will first establish a secure channel (via a shared key k_{uid}) with the enclave via remote attestation. The user encrypts the 4-tuple (uid, pwd, bid, op) with the shared key k_{uid} where pwd is user's authentication password and op denotes the access type (e.g., $\mathsf{op} \in \{\mathsf{read}/\mathsf{write}\}$), and then sends the encrypted request to the storage server, which passes it into the enclave (Step 1). The enclave decrypts the tuple (uid, pwd, bid,op) using k_{uid}, and authenticates the user with the uid and password pwd. If authenticated, it derives the ID bid′ of a block

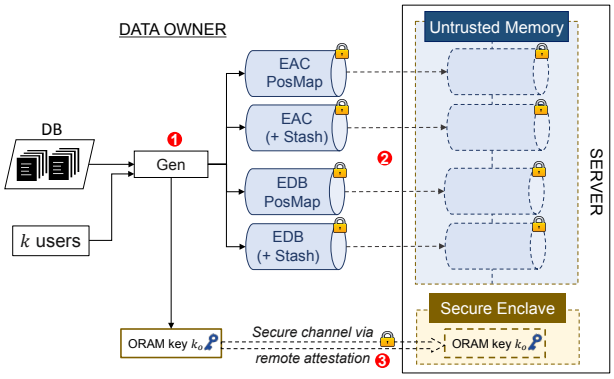

Figure 1: MOSE initialization.

MOSE.Gen(λ, DB, k, N):

1: $k_o \leftarrow \mathcal{E}.\text{Gen}(1^\lambda)$ an initialize $k \times N$ matrix AC ▷ Generate ORAM key k_o
2: EDB $\leftarrow$ Circuit-ORAM.Setup(k_o, DB)
3: AC[uid, bid] $\leftarrow$ {R, RW, $\perp$} for $1 \le$ bid $\le N$ and $1 \le$ uid $\le k$
4: AC′ $\leftarrow$ Split AC into N' blocks each of size $|B'|$ bits
5: EAC $\leftarrow$ Circuit-ORAM.Setup(k_o, AC′)
6: **return** (k_o, EDB, EAC)

Figure 2: Setting up EAC and EDB components in MOSE.

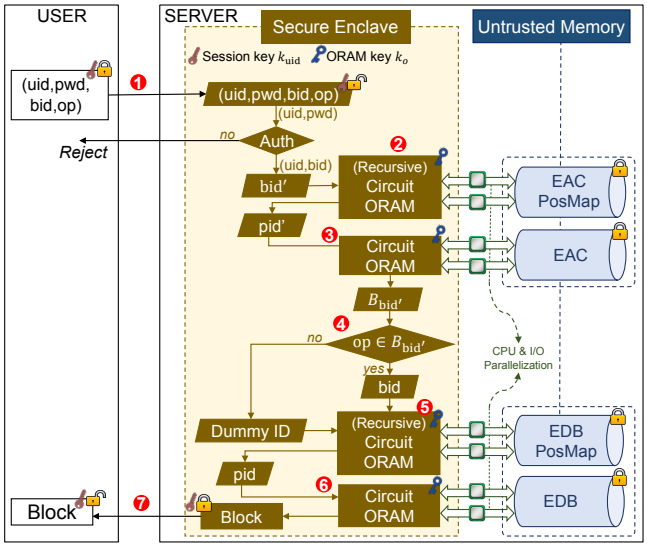

Figure 3: Oblivious access workflow in MOSE.

in EAC containing the permission of uid with bid. The enclave performs recursive Circuit-ORAM accesses on the position map component of EAC to retrieve the path location (pid′) of bid′ in EAC (Step ❷). After that, the enclave performs a Circuit-ORAM access on EAC to retrieve the block bid′ from the path pid′ (Step ❸), and checks whether op is permitted (i.e., op $\in$ bid′) or not (Step ❹). If op $\in$ bid′, the enclave performs recursive Circuit-ORAM accesses on the position map component of EDB to retrieve the path location of bid (Step ❺), and then executes a Circuit-ORAM access on EDB to retrieve the requested block bid from the path (Step ❻). Otherwise, if op $\notin$ bid′, the enclave performs *dummy* (recursive) accesses on both EDB's position map and EDB. Such

MOSE.AccessDB(uid, bid, op, B^*):

1: bid′ $\leftarrow (2N \cdot \text{uid} + \text{bid}) / |B'|$ ▷ N: # DB blocks, $|B'|$: AC block size
2: $B' \leftarrow$ Circuit-ORAM.Access(read, bid′, $\perp$; EAC)
3: perm $\leftarrow$ Get the permission of user uid on bid from B'
4: $v \leftarrow$ ocmp(op, perm) ▷ Check if op = perm
5: did $\xleftarrow{\$} \{1, \ldots, N\}$
6: bid $\leftarrow$ oupt(v, bid, did); $B^* \leftarrow$ oupt(v, B^*, $\{0\}^{|B^*|}$)
7: $B \leftarrow$ Circuit-ORAM.Access(op, bid, B^*; EDB)
8: $B \leftarrow$ oupt(v, B, $\{0\}^{|B|}$)
9: $B' \leftarrow \mathcal{E}.\text{Enc}_{k_{\text{uid}}}(B)$
10: **return** B' ▷ Send B' to uid

Figure 4: Processing user request in MOSE.

MOSE.WriteAC(uid, bid, p):

1: bid′ $\leftarrow (2N \cdot \text{uid} + \text{bid}) / |B'|$ ▷ N: # DB blocks, $|B'|$: AC block size
2: $B' \leftarrow$ Circuit-ORAM.Access(write, bid′, p; EAC)

Figure 5: Update user permission in MOSE.

dummy accesses are necessary in order to prevent the server from learning whether the user request is permitted or not. Finally, the enclave encrypts the accessed block (or dummy data if uid is not permitted) with k_{uid}, and then sends the encrypted data to the user (Step ❼).

Figure 4 presents how our enclave processes the access request algorithmically after the user is authenticated. The algorithm invokes the recursive Circuit-ORAM protocol (Steps 2, 7). Generally speaking, the enclave needs to perform a conditional check to verify whether the user has permission or not (Step 4) and also require a check when the enclave obliviously accesses the database block from the path of the Circuit-ORAM tree. To prevent information leakage from diverging execution on a conditional branch, we implemented the oblivious comparison and oblivious update functions using `CMOV` and `SETE` instructions proposed in [36, 37], which are defined as follows.

- $b \leftarrow \text{ocmp}(x, y)$: It takes as input two values x, y and outputs $b \leftarrow 0$ if $x = y$ or $b \leftarrow 1$ otherwise.
- $z \leftarrow \text{oupt}(b, x, y)$: It takes as input two values x, y and a boolean b, and assigns $z \leftarrow x$ if $b = 1$ or $z \leftarrow y$ otherwise.

Notice that read and write operations (i.e., ReadDB and WriteDB algorithms in Definition 3.1) must incur the same procedure presented in Figure 4 to achieve security. To be complete, Figure 5 presents the detailed WriteAC algorithm defined in Definition 3.1. Notice that this process can only be triggered by the data owner, and the enclave executes this algorithm only after the data owner is authenticated.

5.3 Towards Scalable Oblivious Access

To achieve a scalable performance, we optimize each Circuit-ORAM access on encrypted data components stored in the untrusted memory via secure enclaves and parallelization of computation and I/O accesses. As briefly discussed in §3, the state-of-the-art proxy-based oblivious storage design (TaoStore [40]) pointed out the concurrency limitation of ORAM. That is, any optimization that parallelizes multiple ORAM access must reply such requests sequentially in their arrival order to prevent information leakages via their timings. Although multiple user's requests can arrive and be processed at the proxy simultaneously, the total response time for a user is

equal to the total processing time of its prior requests plus the processing time for the target request, which includes local proxy computation and ORAM network communication delay.

Holding the restriction that all requests have to be replied in a sequential order to ensure security, the only solution to optimize MOSE's performance is to employ parallelization tricks, not for the concurrent ORAM access, but for minimizing the processing time of a single request. With this approach, we can minimize the delay of processing one user request at a time and also make MOSE scalable by increasing its concurrent request processing performance in proportional to the number of available CPU cores.

Another bottleneck that limits the support of concurrent requests is the network bandwidth overhead. The 1 Gbps link between the proxy and the storage server only allows *ten* concurrent access requests in a second, and this number cannot reflect concurrency needs in real-world scenarios. Thanks to MOSE's design that utilizes secure enclaves, the network overhead does not exist in MOSE because the enclave communicate with the storage via a high-speed memory bus, which is a 273 Gbps link for a regular dual-channel DDR4-2133 memory.

Scalable Computation and I/O Access Parallelization. MOSE parallelizes computations as well as I/O accesses in each Circuit-ORAM access by utilizing a multi-core CPU as follows. We split each (real/dummy) block of Circuit-ORAM into multiple chunks, preferably as the number of parallelized threads, and encrypt each chunk separately and independently from one another using parallelizable encryption techniques such as AES-CTR. On performing a Circuit-ORAM access, the enclave spawns multiple threads to access such chunks in parallel. Each thread is responsible to read/write its assigned chunks from/to the server's storage device and also perform encryption/decryption. This parallelization is simple, but more efficient than [9], in which each CPU is assigned to an independent subtree in the ORAM tree structure, which might incur a costly synchronization and unbalanced CPU workloads. Moreover, because it only parallelizes a single ORAM access request, this parallelization is not vulnerable to the asynchronousity attack [40].

A caveat in applying this parallelization is that the meta-data components in the Circuit-ORAM tree is very small in practice (less than 16 B), the parallelization for meta-data computation is not beneficial because initializing threads costs some delay. Therefore in MOSE, we use only one thread to load the encrypted meta-data from the storage disk into the enclave memory space first, and then creating multiple threads to process block data by chunks.

Performance and storage trade-offs in access control structures. In this study, we instantiate AC with matrix, which is the basic structure for access control management. This design decision follows the same scalability principle: optimizing a single request. Although matrix is storage-costly (sized as # of user × # of database files) than more compact alternatives such as linked-list, it allows to get the user permission on a specific database block directly when packaged into the ORAM blocks. Specifically, the cell AC[uid, bid] can be packaged into an ORAM block with ID $\text{bid}' = (2 \cdot \text{uid} \cdot N + \text{bid})/|B'|$, where N is the number of database blocks, and $|B'|$ is the selected AC block size (in bits). On the contrary, with a linked-list structure, each entry for uid stores an array of block IDs that uid can access to (or in reverse). Because the list is different for each user, one might have to scan at least logarithmic number of entries in the list with multiple ORAM executions (e.g., [52]), which will incur a high latency. Moreover, such structure might be vulnerable to information leakage via list size and access timing, which cannot be prevented by ORAM. Padding can mitigate this leakage, but it incurs more delay and is application-specific.

Note that the entire operation of MOSE is orthogonal to the selection of AC structure, so any structure can be utilized. However, for the given scalability goal and security requirements, we decide to sacrifice the storage for the other properties.

6 SECURITY

We present the security definition of standard ORAM as follows.

Definition 6.1 (ORAM security [42]). Let $\vec{o} = (\langle \text{op}_1, \text{idx}_1, \text{data}_1 \rangle, \ldots, \langle \text{op}_q, \text{idx}_q, \text{data}_q \rangle)$ be a sequence of data access requests on encrypted database, where $\text{op}_i \in \{\text{read}, \text{write}\}$, idx_i is logical address to be read/written and data_i being the data at idx_i to be read/written (for $i \in \{1, \ldots, q\}$). Let $\mathbf{AP}(\vec{o})$ be an access pattern observed by the server $\mathcal{S}$ given a data request sequence $\vec{o}$. An ORAM scheme is secure if for any two data request sequences $\vec{o}_i$ and $\vec{o}_j$ of the same length, their access patterns $\mathbf{AP}(\vec{o}_i)$ and $\mathbf{AP}(\vec{o}_j)$ are computationally indistinguishable by anyone but the client.

We present the security notion for an enclave-assisted multi-user ORAM with access control, and show how MOSE achieves them. Our model is inspired from [30, 31, 40] with the following differences. In [30, 31], the security model captures multi-user ORAM and access control *without* a trusted proxy (modeled as a secure enclave in MOSE). In [40], it captures a multi-user ORAM with a trusted proxy over the asynchronous network setting but does not offer access control measures. In MOSE, the enclave resides on an untrusted server and therefore, the network setting does not apply. We consider two main adversary properties including *maliciousness*, where they can actively inject/tamper with the input and *collusion*, where both the user and server can collude with each other.

Access pattern obliviousness: Intuitively, an enclave-assisted multi-user ORAM with access control is secure if the server and an arbitrary subset of users, cannot learn any information regarding the access patterns of honest users aside from what is trivially leaked by corrupted users. We define this notion as follows.

Definition 6.2 (Enclave-assisted multi-user ORAM with access control security). Consider the following experiment between an adversary $\mathcal{A}$, which contains a set of malicious and colluding entities (i.e., users and server) and a challenger C (which works as OClient).

- *Setup:* C generates a database DB with N blocks for k users and initializes two empty lists $\mathcal{UL}$ and $\mathcal{QL}$. It executes $(k_o, \text{EAC}, \text{EDB}) \leftarrow \text{Gen}(1^\lambda, \text{DB}, k, N)$, and returns EAC and EDB to $\mathcal{A}$.
- *Learning phase:*
  - $\underline{O\text{addU(uid)}}$: This oracle adds a user and its secret key to the list of colluding users as $\mathcal{UL} \leftarrow \mathcal{UL} \cup \{\text{uid}, k_S\}$.
  - $\underline{\text{OAccess(op, uid, bid, data)}}$: If op = read, it executes ReadDB(uid, bid; EAC, EDB). Otherwise (if op = write), it executes WriteDB(uid, bid, data; EAC, EDB). It then executes $\text{data}' \leftarrow \text{Response}(k_{\text{uid}}, \text{data})$ and add $o_i = \langle \text{op}, \text{uid}, \text{bid}, \text{data} \rangle$, $\mathbf{AP}(o_i)$ to $\mathcal{QL}$, where $\mathbf{AP}(o)_i$ is the access pattern when executing WriteDB, ReadDB and Response protocols.
- *Distinguish phase:* $\mathcal{A}$ prepares two access requests $o_1 = \langle \text{op}_1, \text{uid}_1, \text{bid}_1, \text{data}_1 \rangle$ and $o_2 = \langle \text{op}_2, \text{uid}_2, \text{bid}_2, \text{data}_2 \rangle$.
  - $\underline{\text{Challenge}(o_1, o_2)}$: $\mathcal{A}$ queries the challenge oracle with two access requests o_1 and o_2 defined above. If $\text{uid}_1 \in \mathcal{UL}$ or

$\mathsf{uid}_2 \in \mathcal{UL}$, C aborts. Otherwise, C flips a random coin $b \leftarrow \{0,1\}$ and executes ReadDB or WriteDB depending on op_b and returns $\mathbf{AP}(o_b)$ to the adversary. The adversary can continue the learning phase with the exception of calling $O\mathsf{addU}(\mathsf{uid}_1)$ or $O\mathsf{addU}(\mathsf{uid}_2)$. $\mathcal{A}$ eventually outputs a bit b' to indicate if $\mathbf{AP}(o_b)$ corresponds to o_1 or o_2.

At the end of the game, the adversary wins if $b' = b$.

Data secrecy: Only the authorized users can learn about the database contents.

Tamper resistance against malicious server: Adversary can not violate data integrity.

Theorem 6.1. *MOSE is secure by Definition 6.2 if the underlying ORAM and IND-CPA encryption scheme are secure.*

Proof. The game starts by running the Gen algorithm to initialize EDB, EAC and the ORAM key k_o. During the learning phase, the adversary can corrupt any user of its choice by simply querying its key. $\mathcal{A}$ can adaptively query the OAccess oracle on access requests of its choice. At some point, $\mathcal{A}$ decides to query the Challenge oracle. $\mathcal{A}$ prepares two access requests $o_1 = \langle \mathsf{op}_1, \mathsf{uid}_1, \mathsf{bid}_1, \mathsf{data}_1 \rangle$ and $o_2 = \langle \mathsf{op}_2, \mathsf{uid}_2, \mathsf{bid}_2, \mathsf{data}_2 \rangle$. Note that to rule out the trivial cases, we require that $\mathsf{uid}_1 \notin \mathcal{UL}$ or $\mathsf{uid}_2 \notin \mathcal{UL}$.

Upon initiating the Challenge oracle, C commits to a random coin $b \leftarrow \{0,1\}$. For a $o_b = \langle \mathsf{op}_b, \mathsf{uid}_b, \mathsf{bid}_b, \mathsf{data}', \mathsf{data}^* \rangle$ submitted to the Challenge oracle, whether op_b is a read or a write operation, two ORAM accesses take place. The first ORAM access is to EAC to determine the permission of uid_b on bid_b. When $(p_b, \mathsf{EAC}') \leftarrow \langle \mathsf{ReadAC}_{\mathcal{A}(\mathsf{EAC})}(\mathsf{uid}_b, \mathsf{bid}_b) \rangle$ is called, EAC is accessed by the underlying ORAM and the output (i.e., p_b) is returned to C. After determining uid_b's permission on bid_b, C performs either a real or dummy ORAM access for data request o_b on EDB. This leads to memory access pattern $\mathbf{AP}(o_b)$ on EDB where the output is returned through $\mathsf{data} \leftarrow \langle \mathsf{Response}(k_{\mathsf{uid}}, \mathsf{data}) \rangle$ to uid_b.

We now analyze the view of $\mathcal{A}$ for the access patterns and transcripts generated through the above accesses. First, both EAC and EDB are encrypted via an IND-CPA encryption scheme at all times. Second, when $(p_b, \mathsf{EAC}') \leftarrow \langle \mathsf{ReadAC}_{\mathcal{A}(\mathsf{EAC})}(\mathsf{uid}_b, \mathsf{bid}_b) \rangle$ is called, $\mathcal{A}$ does not have any view on outputs to C, therefore, it cannot infer any information about p_b. Moreover, since MOSE leverages a secure ORAM, to access EAC, any memory access pattern generated by ORAM is (computationally) indistinguishable by Definition 6.1 [8]. Third, based on the permission p_b, whether $\mathcal{A}$ has to perform a real or dummy access for request o_b on EDB, due to the security of the underlying ORAM, the generated access patterns $\mathbf{AP}(o_b)$ for $b \in \{0,1\}$ is (computationally) indistinguishable by Definition 6.1. Lastly, the output of Challenge to $\mathcal{A}$ is encrypted with an IND-CPA encryption and therefore, indistinguishable for $\mathcal{A}$. In all the above cases, for $\mathcal{A}$ to distinguish between $\mathbf{AP}(o_b)$ based on o_1 and o_2, it has to break the underlying ORAM or IND-CPA encryption. □

Corollary 6.1. *MOSE offers data secrecy and tamper resistance against malicious adversary.*

Proof. Data secrecy in MOSE is based to the IND-CPA property of the underlying encryption scheme. As pointed out in the above proof, EDB and EAC remain encrypted at all the times via an IND-CPA encryption. Moreover, the returned values are also encrypted in the Response algorithm via the underlying IND-CPA encryption.

The tamper resistance of MOSE is based on the underling keyed hash function (e.g., SHA-256) which is used to provide HMAC for each database block. Finally, the integrity of the enforced access control mechanism ensures users cannot access (read/write) that they do not have permission on. □

Blocking side-channel attacks. Although the enclave's data is isolated and encrypted, attackers could indirectly learn about the data via cache [19, 26, 32] or page-fault [21, 49, 55] side-channel attacks. Specifically, the execution that retrieves a block from a read path of ORAM tree or accesses stash could leak information related to data to such side channels. In this regard, MOSE provides defenses against such attacks. On one hand, we access all blocks in a single path and the stash per each ORAM access to prevent cache side-channel attacks. On the other hand, we implement our logic in the enclave using `CMOV` instructions to remove conditional branches that potentially leaks via page-fault side-channel when the enclave verifies the user permission.

7 EVALUATION

Implementation. We implemented MOSE in C/C++ using the Intel SGX SDK v1.7. Our implementation contains approximately 2,982 lines of code for the untrusted modules and 780 lines of code for trusted modules. We leveraged `sgx_aes_ctr_encrypt()` and `sgx_read_rand()` functions in the Intel SGX SDK library, for encrypting ORAM with AES-CTR mode and random number generation (via the `RDRAND` instruction), respectively. We used `pthread` for spawning multiple threads to support parallelism in secure enclave. Remark that we stored the position map components on the untrusted memory in the form of recursive Circuit-ORAM structures. We also stored the stash components on the untrusted memory, which are encrypted and loaded as plaintext only to the enclave memory space chunk-by-chunk (an `ECALL` function handles this).

In the following, we outline the configurations and methodology of our evaluation (in §7.1), and then we evaluate the effectiveness of MOSE in terms of the delay when handling single/multiple client request(s) with/without optimization and the storage overhead.

7.1 Configurations and Methodology

Hardware. We used a commodity desktop supporting Intel SGX, which is equipped with a six-core Intel Core i7-8700K CPU @ 3.70 GHz, 32 GB of dual-channel DDR4-2133 memory, and 4 TB NVMe SSD drive.

Dataset. We constructed a database DB containing 2^{25} (33,554,432) random database blocks of size 24 KB (4 KB × six threads). We assumed basic 2-bit access policies (read and/or write) for 2^{14} (16,384) users on such 2^{25} blocks.

Evaluation Methodology. We evaluated the performance in terms of latency, throughput, and memory usage, by varying the database size, number of cores and ORAM cache level (see §7.2). Afterwards, we applied various optimizations such as position map caching and k-top level caching (see §7.3). We then analyze the effectiveness of MOSE in the multi-user environment, compare its performance with TaoStore [40] under various database sizes. (see §7.4). We did not compare with other proxy-based techniques [4, 40, 45] because their design is insecure against asynchronous timing attack (see §8). We also did not compare MOSE with non-proxy ORAM primitives [5, 8] because they incur high communication/computation

overhead due to some cryptographic operations. For scalability testing, we created a number of virtual users that send concurrent access requests to MOSE with 50 ms network latency.

Configurations. We setup the following system parameters.

• *MOSE:* We selected standard parameters for Circuit-ORAM: bucket size $Z = 2$ with deterministic eviction and stash size $|S| = 80$. To exploit parallelism on accessing the ORAM structure from the NVMe disk drive, we divided each block of EDB into n_t 4 KB chunks, where $n_t = 6$ being the number of threads for parallelism. We instantiated AC with a matrix for access control management of 2^{14} users on 2^{25} data blocks. We divided AC into 12- KB blocks, and built recursive Circuit-ORAM trees for such blocks and their position map. Similar to EDB, we divided each EAC block into n_t chunks, and each chunk is of size 2 KB. For the recursion, we selected the compression ratio $r = 256$.

• *TaoStore [40]*: We launched a simulation experiment for TaoStore with a conservative approach. Our virtual TaoStore used Path-ORAM because it was used by default in [40] and is the most efficient ORAM for (networked) client-server applications. We used the Path-ORAM standard parameters: $Z = 4$, $|S| = 80$ [47]. We selected 1 Gbps of network throughputs with 10 ms latency for simulating the execution of the Path-ORAM on the proxy. We excluded all execution delays at the storage server and the proxy such as I/O access, decryption/encryption, thread synchronization, etc. and only simulated the network delay of transmitting Path-ORAM paths caused by executing Path-ORAM protocols (in parallel) over the network between the proxy and the server. Our logic behind this experiment is that the network delay is inherent so it must be included, and adding any of the implementation to the server and proxy will certainly incur more delay on the execution side. Any actual implementation of TaoStore will involve more delay than this simulation.

7.2 Single Request Processing Time

Base-case performance. We first present MOSE's response time in handling a single user request for various database sizes from 6 GB to 768 GB, while not applying any optimizations such as k-level cache, etc. (see Table 1). The size of database will affect the path length, so our enclave will read more data from the ORAM, and therefore will result in more delay. In the base-case design, the average total delay of MOSE is from 9.21 ms to 28.05 ms. In MOSE, the accesses on EAC and EDB can be slightly pipelined. That is, right after finishing the access on EAC from one request, MOSE can issue the access on EAC of the next request while processing EDB access of the previous request. This pipelining strategy allows MOSE to achieve the throughputs ranging from 200 ops to 53 ops for database sizes from 6 GB to 768 GB, respectively.

Note that these numbers do not reflect the optimized performance. Next, we will further analyze factors that affect the total delay in MOSE, and then optimize them.

Delay Breakdown. We dissect the cost of a single request to understand the factors that affect MOSE's performance. As illustrated in Figure 6, MOSE has two major sources of delay: (i) the I/O accesses between enclave and disk incurred by Circuit-ORAM via encrypted read/write operations; (ii) the secure computation (e.g., decryption/re-encryption/oblivious update operations) in the enclave. In the breakdown, we can observe that most delays in MOSE are caused by four different I/O accesses: The position map for the access control, the access control data, the position map for the database, and the database block. For the database accesses, the delay is caused by their tree ORAM structure, where each read/eviction operation requires accessing $O(\log N)$ blocks located in random positions on the disk. For the position map accesses, 92% of its delay is caused by I/O access (uses only 8% of time for computation) because it is stored in the recursive ORAM structures, which require multiple access rounds. In the next section, we will evaluate optimization techniques applied to MOSE to reduce this I/O delay.

Table 1: Delay of base-case MOSE in processing one user request, for various database sizes.

DB Size	Acc. Control (ms)		Database (ms)		Total (ms)
	pos_{EAC}	EAC	pos_{EDB}	EDB	
6 GB	1.46	2.77	1.43	3.55	9.21
12 GB	1.64	2.78	1.54	3.72	9.62
24 GB	2.36	3.49	2.55	4.63	13.03
48 GB	2.51	3.79	3.46	6.29	16.05
96 GB	2.56	3.92	4.94	7.53	18.95
192 GB	2.78	4.13	6.05	8.7	21.66
384 GB	3.52	4.84	7.06	9.61	25.03
768 GB	3.67	5.14	7.97	11.27	28.05

7.3 Optimized MOSE

To reduce the single-request delay, we implemented various optimization techniques including caching and parallelization to minimize the I/O access and computation delays.

Optimizing I/O delay. We first cache the entire position map, stash, and metadata, all of which are required to make an ORAM access, in the main memory. For instance, the size of position map is only 5.6 MB for a 12 GB DB and 47.8 MB for a 96 GB DB, while the size of stash and metadata are approximately 400 MB. Due to their relatively small sizes, maintaining them in the main memory incurs a low overhead. The second and third bars in Figure 7 illustrate the outcome of this optimization. Caching the entire position map reduces the I/O delay from 14.93 ms to 8.03 ms (46.21% reduction). Moreover, caching the stash and meta-data components reduces I/O delay from 8.03 ms to 3.01 ms (62.51% reduction) on top of the position map caching. In summary, applying the caching mechanisms reduces MOSE's I/O delay from 14.93 ms to 3.01 ms, which is 79.83%

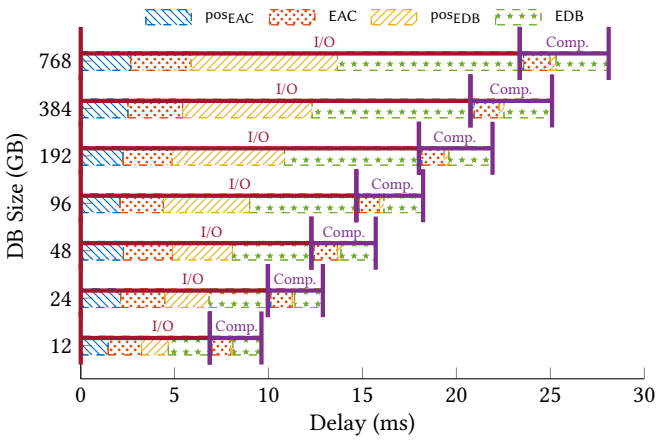

Figure 6: Delay breakdown of base-case MOSE on a single user request for various database sizes.

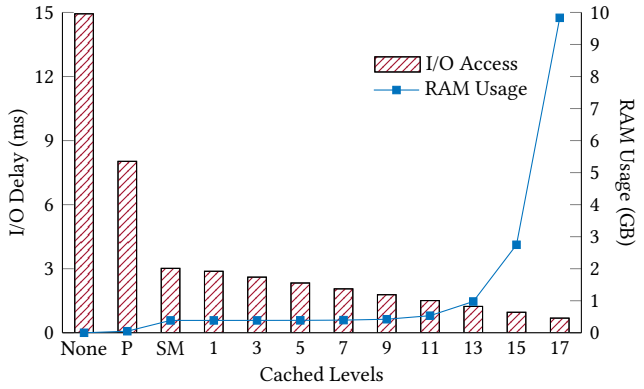

Figure 7: The impact of caching on the I/O delay and memory usage of MOSE. "None" denotes no caching; "P" denotes caching the position maps; "SM" denotes caching the stashes and meta-data; x s-axis represents the catching levels of ORAM trees of EAC and EDB.

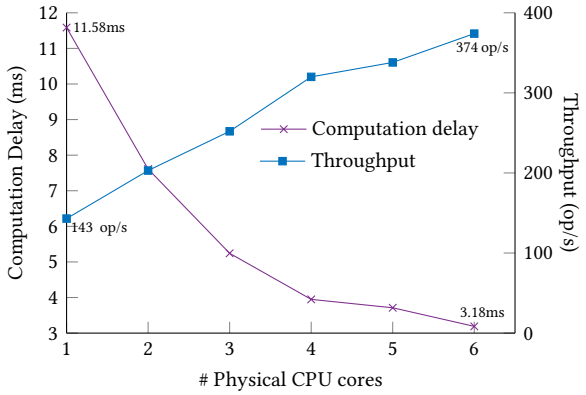

Figure 8: The impact of number of cores on the computation efficiency and scalability of MOSE.

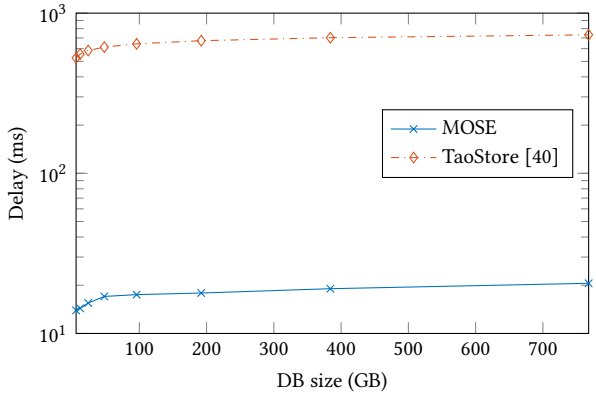

Figure 9: The performance of MOSE *vs.* TaoStore with varied DB sizes under 1 Gbps network throughput.

delay reduction (see the third bar of Figure 7), while incurring ≈ 500 MB of additional memory usage.

We also implemented the k-top caching strategy proposed in [29], which caches the first k levels of the EDB and EAC structures, to reduce I/O delay in ORAM access. The latter bars (4^{th}+) of Figure 7 illustrates the outcome of the caching. Increasing the number of cached levels reduces the I/O delay with the cost of a higher memory usage, however, note that performance gain increases linearly (by reading $H - k$ blocks where H is the tree height), and memory overhead increases exponentially (caching $2^{(k+1)}$ blocks for k-top caching). In this regard, we should set a practical limit of cached level by which the server can accommodate its memory overhead. Overall, we can reduce the I/O delay of MOSE from 14.93 ms to 0.68 ms by using around 10 GB of additional memory. As shown in Figure 7, caching around 50-70% levels of EDB and EAC provides a reasonable memory and I/O delay trade-off (e.g., 11–13 levels with 1 GB RAM usage for 96 GB DB).

Optimizing computation delay. MOSE also leverages the parallelism in a multi-core CPU to speed up encryption/decryption operations in the enclave, and this optimization makes MOSE scalable. The purple line in Figure 8 illustrates the impact of utilizing multiple CPU cores in MOSE's computation logic. The performance of MOSE increases in proportion to the number of CPU cores that MOSE uses. The actual gain by a multi-core CPU is slightly lower than linear increment, e.g., using 6 physical cores improved around 4× of the performance. This is because creating and assigning multiple threads into the corresponding physical CPU core costs a fixed and remarkable overhead.

Overall delay after optimization. When all optimization techniques above are applied, MOSE will take 3.74 ms in total to process an access request on a 24 KB block in a 96 GB DB, which consists of 0.56 ms I/O delay and 3.18 ms computation delay. This allows MOSE to process around 394 op/s with pipelining strategy.

Scalability of MOSE. MOSE's performance is scalable, i.e., MOSE performs better with a more number of CPU cores on the server. The blue line in Figure 8 illustrates how the number of concurrent users for supporting less than one second delay increases as the number of cores that MOSE used for the experiment increases.

7.4 Multi-user Concurrent Access Performance

MOSE with practical concurrency. We ran an experiment on a virtual network that assumes 50 ms for the user network latency, and it showed that on a 96 GB DB, MOSE can serve 374 concurrent requests, each accessing a 24 KB block without incurring more than a second delay for 374 users. In case if the server's service-level agreement (SLA) is to ensure average expected delay for each user to be one second, MOSE can serve around 788 concurrent requests.

MOSE *vs.* TaoStore. We compare the performance of MOSE with TaoStore [40] with varied database sizes. We considered *ten* concurrent requests[1] as originally presented in [40] for a fair performance comparison. For the experiment, we simulated the performance of TaoStore according to 24 KB block size and varied database sizes ranging from 1 GB to 1 TB. Figure 9 presents the delay of MOSE and TaoStore. Note that since our database and block size are larger than that of [40], the reported delay of TaoStore in Figure 9 is higher than what is originally presented in [40]. Given TaoStore is equipped with a highly dedicated network (i.e., 1 Gbps throughput), MOSE is still around 34.80–37.8× faster than TaoStore where MOSE

[1] The notion of concurrency in TaoStore is limited by the communication bandwidth between the server and the trusted proxy. The reason for having maximum ten concurrent users in their evaluation is mainly because ten concurrent requests will max-out the bandwidth. Having more request than ten at the same time will drastically degrade its performance, as stated in their paper.

(with optimization) achieved (on average) 13–21 ms delay for each request, compared with 525–731ms (on average) in TaoStore.

7.5 Storage Overhead

MOSE stores four encrypted components in the server's untrusted storage device: EAC and EDB, and their position maps. All of them are in the form of Circuit-ORAM tree structures, which incur a constant ($\approx 2\times$) storage blowup. Specifically, for the 96 GB DB with 2^{14} users, EDB and EAC cost approximately 206.8 GB and 103.7 GB, respectively. The position map components cost approximately 48.8 MB (1/2000 of DB size).

8 RELATED WORK

Oblivious Random Access Machine (ORAM) was first explored by Goldreich et al. [18] for software protection, and then has been intensively studied in the context of data outsourcing (e.g., [42, 44, 45, 47]). Shi et al. [42] proposed the tree structure for ORAM, which was extensively used in recent ORAM constructions (e.g., [22, 39, 47, 51]). Tree-based ORAM schemes (without server computation) feature $O(\log N)$ client-server communication overhead, and $O(1)$ (e.g., [51]) or $O(\log N)$ [47] stash size.

Non-proxy ORAMs. Several Parallel-ORAM schemes have been introduced (e.g., [6, 8, 9, 35]), which can be used to enable multi-user concurrent access. These constructions feature large block size (e.g., [8]), or add poly-logarithmic communication blowup atop standard ORAM (e.g., [6, 9, 35]). Maffei et al. proposed several access control-supported ORAM schemes (e.g.,[30, 31]). The authors also gave an insightful computation lower bound of $\Omega(N)$ for the composition of access control and ORAM against active adversaries.

Proxy-based asynchronous ORAMs. ObliviStore [46] and its extended multi-server version [44] were among the first to exploit a trusted proxy for concurrent oblivious access in the multi-user setting. In these systems, the proxy schedules concurrent user's requests and then, executes Partition-ORAM with the storage server [4, 40, 44, 45]. Since ObliviStore only parallelizes requests on different blocks, but *sequentializes* same-block requests, it leaks timing information. This allows an adversary to distinguish access patterns from multi-users. To seal this leakage, CURIOUS framework [4] also parallelizes same-block requests by invoking one actual ORAM operation along with several dummy ORAM operations between the proxy and storage server. Although this design enables CURIOUS to be *fully asynchronous* (i.e., one request can be processed immediately without waiting for the previous requests to be finished), Sahin et al. [40] showed that it still leaks timing information. Specifically, because dummy ORAM operations do not return the actual block being requested to the proxy, concurrent requests on the same block cannot be answered until the real one is finished. Given that the adversary can observe all network traffic coming from the proxy, and even reschedule the network package delivery from the storage server, it can learn the timing difference in replying same-block requests *vs.* different-block requests of the proxy. Another limitation is that, they rely on Partition-ORAM, which incurs costly communication overhead due to the background eviction and high proxy storage overhead (i.e., $O(\sqrt{N})$).

To seal timing information leakage, Sahin et al. [40] indicated that the proxy must reply users' requests *sequentially* according to their arrival order. The authors proposed TaoStore, which implements two modules called *Processor* and *Sequencer* at the proxy, where the former executes a standard ORAM (i.e., Path-ORAM [47]) with the storage server in parallel via multiple threads, while the latter is to reply users' requests sequentially. Once a thread obtains data from the storage server due to ORAM operation, *Processor* will lock the proxy's local memory and synchronize it with the fetched data, and then transfers the desired block to the *Sequencer* for user reply. TaoStore is also more efficient than previous designs because it employs Path-ORAM [47].

Using secure hardware to make oblivious access more practical has been explored in the literature [1, 3, 23, 28, 41, 43]. Unlike MOSE, most these techniques either focus on the single-user setting (e.g., [1, 41]) or harness a custom hardware such as FPGA (e.g., [29, 34]). One of the most most relevant system to ours is Shroud [28], which harnesses multiple commodity CPUs to boost the ORAM performance in serving concurrent multi-user requests. However, we note that Shroud focuses on the optimization in a very large-scale server, where it needs thousands of CPUs (around 8,192) for parallelization, while MOSE scales linearly from one core to many. Similar to our MOSE, Obliviate [1] and POSUP [23] also made use of Intel-SGX to reduce the network bandwidth of ORAM. The objectives of these works is different than ours, where they focused on oblivious file system and searchable encryption, instead of multi-client setting and access control enforcement. Another relevant work is [13], which harnesses Intel SGX to enforce access control policy and encryption services. However, it does not provide the access pattern obliviousness like MOSE.

9 CONCLUSION

We presented MOSE, a practical multi-user oblivious storage with access control that enables secure and efficient data outsourcing on the cloud. The adaptation of secure enclaves for multi-user oblivious storage enables a new opportunity to overcome limitations of existing approaches. As a part of this opportunity, MOSE has demonstrated that using secure enclave can eliminate the bandwidth bottleneck and achieve scalability, access control and resiliency against active adversaries simultaneously. For the future direction, we would like to pave a way for scaling multi-user oblivious storage to production levels so that the world can benefit from its high security assurances. An expected, imminent adoption of secure enclaves in public cloud will speed up this goal realization.

ACKNOWLEDGMENTS

This work is supported by the NSF CAREER Award CNS-1917627.

REFERENCES

[1] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious File System for Intel SGX. (2018).
[2] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. 2014. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*. Springer, 131–148.
[3] Dmitri Asonov and Johann-Christoph Freytag. 2002. Almost optimal private information retrieval. In *International Workshop on Privacy Enhancing Technologies*. Springer, 209–223.
[4] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. 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*. ACM, 837–849.
[5] Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir. 2017. Multi-client oblivious ram secure against malicious servers. In *International Conference on Applied Cryptography and Network Security*. Springer, 686–707.
[6] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2016. Oblivious parallel RAM and applications. In *Theory of Cryptography Conference*. Springer, 175–204.

[7] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM CCS*. ACM, 668–679.

[8] T-H Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *Theory of Cryptography Conference*. Springer, 72–107.

[9] Binyi Chen, Huijia Lin, and Stefano Tessaro. 2016. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography Conference*. Springer, 205–234.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. http://eprint.iacr.org/2016/086.pdf.

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

[13] Judicael B Djoko, Jack Lange, and Adam J Lee. 2019. NEXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-side SGX. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 401–413.

[14] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 765–782.

[15] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. 2015. *Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM.* Technical Report. IACR Cryptology ePrint Archive, Report 2015, 1065.

[16] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 386–408.

[17] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[18] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 182–194.

[19] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2.

[20] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 162–168.

[21] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*. 299–312.

[22] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. 2017. S3ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 491–505.

[23] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. 2019. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proceedings on Privacy Enhancing Technologies* 2019, 1 (2019), 172–191.

[24] Intel Corporation. 2013. Intel Software Guard Extensions Programming Reference (rev1). 329298-001US.

[25] Intel Corporation. 2014. Intel Software Guard Extensions Programming Reference (rev2). 329298-002US.

[26] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*. 16–18.

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

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

[29] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 311–324.

[30] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2015. Privacy and access control for outsourced personal records. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 341–358.

[31] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2017. Maliciously secure multi-client oram. In *International Conference on Applied Cryptography and Network Security*. Springer, 645–664.

[32] Urs Müller. 2017. Software Grand Exposure: {SGX} Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies,{ WOOT} 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX.

[33] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 644–655.

[34] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. 2017. Hop: Hardware makes obfuscation practical. In *24th Annual Network and Distributed System Security Symposium, NDSS*.

[35] Kartik Nayak and Jonathan Katz. 2016. An Oblivious Parallel RAM with O (log2 N) Parallel Runtime Blowup. *IACR Cryptology ePrint Archive* 2016 (2016), 1141.

[36] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors.. In *USENIX Security Symposium*. 619–636.

[37] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution.. In *USENIX Security Symposium*. 431–446.

[38] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptology ePrint Archive* 2014 (2014), 997.

[39] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious ram in secure processors. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 571–582.

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

[41] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. 2018. ZeroTrace: Oblivious memory primitives from Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*.

[42] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O ((logN) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. Springer, 197–214.

[43] Sean W. Smith and David Safford. 2001. Practical server privacy with secure coprocessors. *IBM Systems Journal* 40, 3 (2001), 683–695.

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

[45] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 253–267.

[46] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

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

[48] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

[49] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association.

[50] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[51] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 850–861.

[52] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 215–226.

[53] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[54] Peter Williams, Radu Sion, and Alin Tomescu. 2012. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 977–988.

[55] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (Oakland '15)*. IEEE, 640–656.

[56] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium (USENIX Security 16)*. 707–720.