Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset

Thang Hoang* Muslum Ozgur Ozmen* Yeongjin Jang* Attila A. Yavuz*

Abstract

Ability to query and update over encrypted data is an essential feature to enable breach-resilient cyber-infrastructures. Statistical attacks on searchable encryption (SE) have demonstrated the importance of sealing information leakages in access patterns. In response to such attacks, Oblivious Random Access Machine (ORAM) has been proposed. However, the composition of ORAM and SE is extremely costly in client-server model, and this poses a critical barrier towards its practical adaptations.

In this paper, we create a new hardware-supported privacy-enhancing platform called as *Practical Oblivious Search and Update Platform* (POSUP), which enables oblivious keyword search/update operations on extremely large datasets with a high efficiency. We harness Intel SGX to realize highly optimized oblivious data structures for oblivious search/update purposes. We implemented POSUP and evaluated its performance with Wikipedia dataset containing $\geq 2^{29}$ keyword-file pairs. Our implementation is highly efficient, where it takes 1ms to access a 3 KB block with Circuit-ORAM. Our experiments have shown that POSUP offers up to 70× less end-to-end delay and 100× reduced bandwidth consumption, compared with the traditional ORAM-SE composition without secure hardware. POSUP is also at least 10× faster for up to 99.5% fraction of keywords to be searched, compared with existing Intel SGX-assisted search platforms.

1 Introduction

Data privacy and utilization dilemma is a common problem in various applications including, but not limited to, data outsourcing and breach-resilient systems. Recent data breach incidents targeting online applications (e.g., Apple iCloud, Equifax) have shown the importance of protecting data confidentiality on the untrusted cloud environment. A naive approach is to leverage standard encryption techniques such as AES. Unfortunately, such techniques prevent the user from performing even simple queries (e.g., search or update) on encrypted data, and thereby, diminishing the data utilization.

Searchable Encryption (SE) has been proposed to offer data confidentiality and search/update functionalities simultaneously. This is achieved by creating an encrypted index (EIDX), which represents the keyword-file relationships in encrypted files (EDB), and both are outsourced to the cloud. One line of SE research focuses on designing new SE schemes (e.g., [14, 52, 64, 41]) with provable security that offer various trade-offs in terms of security, functionality and efficiency. The other line of research aims to enable encrypted queries compliant to legacy infrastructure such as databases management system (e.g., mySQL, mongoDB) [33, 45, 56, 4, 1, 2, 55, 57]. Despite their merits, these techniques leak access patterns, which results in statistical analysis attacks demonstrated in [40, 48, 77, 13, 51, 31, 13, 58].

*Oregon State University, {hoangmin, ozmenmu, yeongjin.jang, attila.yavuz}@oregonstate.edu

Oblivious Random Access Machine (ORAM) [28] can hide access pattern, and therefore, seal the leakages in SE. Unfortunately, because ORAM is bandwidth-heavy, using ORAM for SE is extremely costly in the standard client-server model [35, 50, 7]. To reduce this communication overhead, recent studies have investigated the support of ORAM with secure hardware [49, 23, 60]. The initial studies designed custom hardware (e.g., FPGA) to enhance ORAM performance, and therefore, they might not be easily integrated into commercial server systems with legacy architecture [3]. With the advent of widely available trusted execution environment on commodity hardware, the deployment of such hardware-supported cryptographic primitives becomes more feasible. Intel SGX [37, 38, 17] has been explored to enable efficient cryptographic operations such as oblivious memory access primitives [61, 3] and functional encryption [22].

Our Objective. The goal of this paper is to take the ORAM supported by the commodity secure hardware into a next level, in which we develop oblivious data structures using Intel SGX to offer practical oblivious search/update operations on very large datasets with boolean queries. We implemented our techniques to show that they are significantly more efficient than existing alternatives.

1.1 Motivation

We elaborate some of the key challenges towards enabling oblivious search and update operations on extremely large encrypted outsourced data as follows.

• *Limitation of ORAM-SE Composition in Standard Client-Server Model:* Due to the ORAM logarithmic bandwidth blowup, the performance of ORAM and SE composition in the standard network setting (depicted in Figure 1) was shown to be extremely inefficient [50, 35, 7]. Naveed et al. [50] conducted an analytical analysis, and concluded that this combination is worse than streaming the entire outsourced data for some keyword distributions. Hoang et al. [35] performed real experiments on the cloud environment, and further demonstrated the inefficiency of a direct ORAM and SE composition in the client-server model. Our experiment in this paper further confirms that the ORAM and SE composition incurs extremely high delays for a large dataset with standard network setting. Our experiment in §6 have showed that this technique incurs one to two magnitude of times client-server bandwidth blowup for both keyword search and update operations.

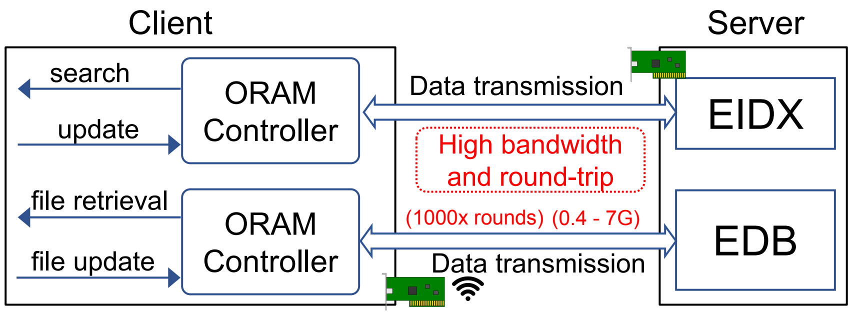


Figure 1. Limitation of ORAM-SE composition in standard client-server model over network.

• *Limitation of Existing Hardware-Supported Oblivious Search Platforms*: Existing systems [24, 69] require the secure hardware (e.g., Intel SGX) to process the entire outsourced data (e.g., encryption and decryption) for each search query in order to completely hide the access pattern (see Figure 2). However, this approach has limitation while dealing with a large amount of outsourced data because its cost grows *linearly* with the data size. In this paper, we show that this strategy (see §6) incurs a significant delay, where it takes around 150 seconds to search for even one file in gigabytes of outsourced data. Moreover, these techniques did not fully investigate the update operation, which is an essential feature of data-outsourcing applications nowadays.

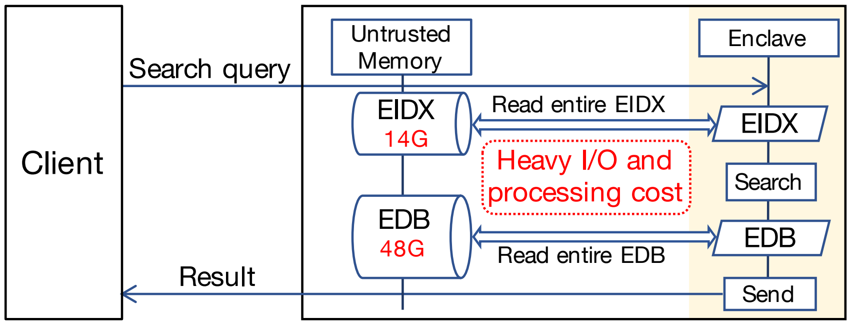


Figure 2. Limitation of existing oblivious search platforms with Intel SGX [24, 24].

ZeroTrace [61] proposed efficient oblivious memory primitives by harnessing recursive Circuit-ORAM with Intel SGX. This system is efficient for generic oblivious access purposes, since it does not require storing the position map and uses Circuit-ORAM with a less circuit complexity compared to Path-ORAM.

1.2 Our Contributions

In this paper, we design a new hardware-assisted privacy-enhancing platform that we refer to as *Practical Oblivious Search and Update Platform* (POSUP). POSUP enables oblivious (multi/boolean)-keyword search and update operations on very large datasets in a much more efficient and practical manner compared with existing techniques.

Our system design is inspired from ZeroTrace [61], where we synergize SGX-supported ORAM with Oblivious Data Structure (ODS) proposed in [74], to enable oblivious keyword search and update operations on encrypted data. This synergy (*i*) addresses the network bandwidth and communication hurdles of ORAM-SE in the client-server setting; (*ii*) eliminates the cost of processing the entire data inside Intel SGX as in [24, 69]; and more importantly, (*iii*) allows to operate on a very large outsourced data without being restricted by Intel SGX memory. This composition also enables efficient oblivious keyword update capacity that was not fully explored in the previous works (e.g., [50, 24]). To the best of our knowledge, POSUP is the first oblivious search and update platform that harnesses secure hardware with ORAM and ODS simultaneously. We further outline our contributions as follows:

1. We construct ODS instantiations for EIDX and EDB by harnessing Intel SGX with Path-ORAM [67] and Circuit-ORAM [72]. POSUP enables various query types such as single-keyword, multi-keyword and boolean queries. Moreover, we propose an optimization strategy that exploits special characteristics of ODS to achieve a highly efficient oblivious update.

2. We implemented POSUP and evaluated its performance on commodity hardware with a very large dataset (e.g., full-size Wikipedia English corpus) consisting of hundred millions of keyword-file pairs and millions of files. Our implementation is highly efficient, where it only takes 1ms to obliviously access a 3 KB block with SGX hardware. Our experimental results showed that, POSUP is highly efficient, which achieves a low end-to-end delay compared with the existing solutions:

• *ORAM-ODS-SE composition in standard client-server model (without secure hardware):* We use traditional ORAM-SE with ODS for a fair comparison with POSUP. This still incurs the network bandwidth and client-server communication round-trip blowups to be 100× and 1000×, respectively, compared with POSUP. As a result, the end-to-end delay of POSUP is *two orders of magnitude* lower than that of this technique for both keyword search and update (see §6 for detail experiments). Our experimental results also confirm the insights given by Naveed et al. [50], where the composition of ORAM and (D)SSE in the standard client-server setting has been shown to be extremely costly.

• *Processing entire data in Intel SGX:* POSUP requires much less amount of data ($O(\log N)$) to be processed in the enclave, compared with existing Intel SGX-assisted search platforms (e.g., [24, 69]), where the entire dataset ($O(N)$) is processed for each query. This results in POSUP to achieve at least 10× lower end-to-end delay than this technique for up to 99.5% fraction of keywords that can be searched (see §6). If the number of searched files is small (e.g., < 16), POSUP can be 100× faster. Moreover, POSUP enables oblivious update, which is not fully investigated in existing SGX-assisted platforms (e.g., [24]).

• *Putting Hardware-Supported ORAM in Effect:* We take the concept of hardware-supported ORAM primitives into the next-level, wherein we develop oblivious data structures and routines with optimizations to provide search and update functionalities on very large data, which was not investigated by existing hardware-supported memory primitives (e.g., ZeroTrace). Our implementation will be open-sourced at

www.github.com/thanghoang/POSUP

2 POSUP Overview

We first outline our objective, and then describe our system model followed by our threat model.

Objective. POSUP's objective is to utilize a public cloud server, which is equipped with a commodity secure hardware, as a secure storage that supports search and dynamic update operations over very large encrypted datasets. To this end, POSUP aims at deploying a practical oblivious encrypted search and update platform to guarantee data confidentiality, no access pattern leakage during search and update operations, along with a trusted execution. Specifically, to defeat attacks against data confidentiality and access pattern leakages, POSUP creates a commodity hardware-supported ORAM and Oblivious Data Structure (ODS) platform, which enables oblivious search and updates efficiently, even for very large datasets with the support of boolean queries. To defeat attacks against the server's execution logic, POSUP runs its ORAM and ODS controller in the enclave protected by Intel SGX.

2.1 System Model

We first describe our system composition and then summarize our POSUP workflow.

System Composition. Figure 3 illustrates components of POSUP and its composition: A client, an untrusted server, and a trusted enclave on the server.

A client (the box on the left side) is a remote entity who generates and manages recursive ORAM and ODS components on the untrusted server via an encryption key k_o. After initializing the system, the client can send a query to update data on the server or to search data and then receive the search results from the server.

The server is comprised of two parts, an untrusted server and a trusted enclave. (i) The untrusted server provides a storage for recursive ORAM and ODS data structures. (ii) The enclave is a trusted part of the server and executes ORAM and ODS controller (while protected by Intel SGX). On behalf of

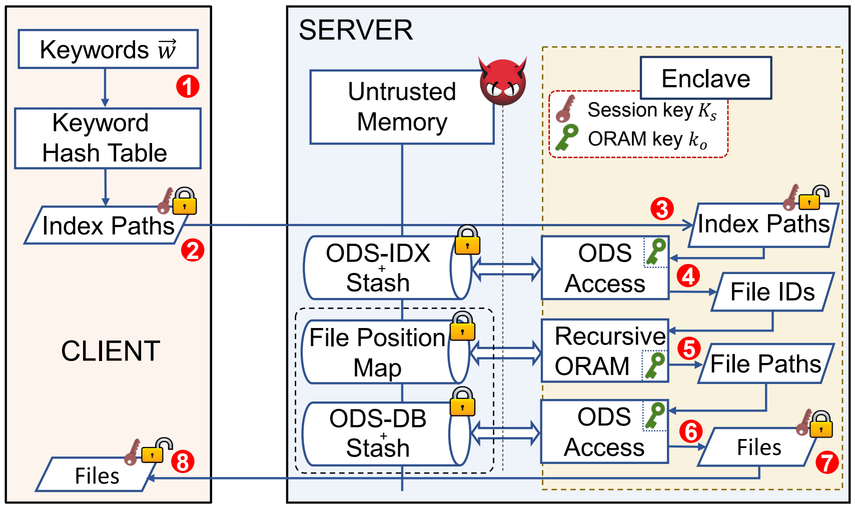


Figure 3. An overview of POSUP workflow. *Initialization*: Oblivious Data Structures (ODS) ODS-IDX and ODS-DB for the encrypted index and encrypted files, respectively, are generated with an ORAM key k_o as described in §4.1, and transferred to the server. Before accessing ORAM on the server, the client runs the remote attestation protocol of Intel SGX to exchange a session key K_s with the enclave. The oblivious search and update after this initialization step are performed as follows: ❶ Given a (search/update) query, the client encrypts index paths corresponding to a keyword with the session key K_s, ❷ and then sends these encrypted data to the enclave (through the untrusted server). ❸ Upon receiving the query, the enclave decrypts the query and ❹ accesses ODS using the ORAM key k_o on the oblivious encrypted index (ODS-IDX) to get file IDs that match with the query. ❺ The enclave determines the location of file IDs in oblivious encrypted files ODS-DB by executing recursive ORAM with ORAM key k_o to file position map structure. ❻ The enclave accesses ODS on ODS-DB with k_o to retrieve files associated with the query. ❼ The enclave encrypts retrieved files with K_s before sending them to the client. ❽ The client recovers encrypted files with K_s.

the client, the enclave performs all oblivious search/update operations upon the client's request on the encrypted data structures stored on the untrusted server. To do this, the enclave receives the encryption key k_o from the client via a secure channel at the initialization.

POSUP Workflow. Figure 3 outlines the overview of oblivious search and update in POSUP.

• *Initialization*: A client first performs the remote attestation of an enclave (provided by Intel SGX), which is running on the untrusted server. This attestation step not only verifies that the program running in the enclave is intact but also exchanges cryptographic keys (a session key K_s) to establish a secure (encrypted) channel between the client and the enclave. After establishing a secure channel, the client also sends a key k_o, which will be used for ORAM operations, to the enclave.

Upon receiving the key, the enclave on the server will initialize encrypted data structures. Our POSUP is comprised of two main encrypted data structures: ODS-IDX, which is an encrypted index that represents keyword-file relations, and ODS-DB, which stores encrypted files. Both data structures are stored in the server's untrusted memory. We employ ODS techniques proposed in [73] to instantiate ODS-IDX and ODS-DB. This data structure initialization step only happens at the first connection. In other words, to perform search and update operations, the client only requires to exchange a session key (K_s) with the enclave.

• *Oblivious Queries/Updates*: (❶-❸) Given a query with keywords $\vec{w}$, the client finds the index path of each keyword from the keyword hash table[1], encrypts all of them with K_s, and sends to the enclave through untrusted server. The enclave decrypts index paths for this query, which will be used to conduct ODS accesses on encrypted data. (❹-❼) POSUP harnesses ORAM controller to perform ODS accesses on ODS-IDX ODS-DB, respectively. Because the enclave is already given index paths for ODS-IDX, it can perform non-recursive ORAM to ODS-IDX to obtain target file identifiers from the untrusted server. It then performs recursive ORAM on the file position map to retrieve the location (path) of file identifiers in ODS-DB, and It then retrieves the corresponding files from ODS-DB obliviously. Finally, the enclave encrypts retrieved files with K_s and sends them to the client, who can decrypt them with K_s.

All these strategies enable us to achieve oblivious keyword search/update operation in a more efficiently than processing the entire ODS-DB and ODS-IDX in the enclave as suggested in [24, 69]. Our system is also more efficient than the direct application of existing SGX-ORAM memory primitives [61] because we harness ODS technique for both ODS-DB and ODS-IDX, which reduces the number of recursive calls when executing an oblivious keyword search/update query.

2.2 Threat Model

We build POSUP based on the following assumptions as its threat model. On the trusted side, we trust the client logic and the ORAM controller logic that runs in the enclave. The client securely stores the ORAM encryption key k_o and only sends it to the enclave after establishing a secure channel between the enclave and the client (therefore the server cannot obtain this key). To establish such a secure channel, we rely on the remote attestation protocol provided by Intel SGX, thus, we trust Intel as a trusted third party.

We assume the server is untrusted except for the enclave. 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 an enclave because Intel SGX isolates and applies encryption to the enclave's memory space using hardware mechanisms.

Side-channel attacks against Intel SGX. Intel SGX does not come without limitation; it suffers from various side-channel attacks in cache access [32, 47, 10, 30], memory access [76, 11], registers [46], etc [75]. POSUP does not aim to defeat all side-channel attacks against Intel SGX, which is an impossible goal; instead, we try to build POSUP as a best-effort approach to defeat known side-channel attacks against Intel SGX. Please refer to §4.3 to see how we defeat such side-channel attacks.

3 Building Blocks

3.1 ORAM

ORAM was first proposed by Goldreich et al. [29] in the context of software protection against piracy, and then, have been studied in the context of data outsourcing. ORAM security can be defined as follows.

Definition 1 (ORAM security [67]). *Let $\vec{o} = (\mathsf{op}_i, u_i, \mathsf{data}_i)_{i=1}^{q}$ be a data request sequence, where $\mathsf{op}_i \in \{read(u_i, \mathsf{data}_i), write(u_i, \mathsf{data}_i)\}$, u_i is the logical address to be read/written and data_i is the data at u_i to be read/written. Let $\boldsymbol{AP}(\vec{o})$ be an access pattern observed by the server given a data request sequence σ.*

[1]Since keyword universe is arbitrarily large, the client must maintain a data structure to uniquely match each keyword to an index for a given dataset. We opt for a hash table in our implementation to achieve constant access time (in average). The client constructs the hash table during the initialization from files and maintains it later.

```
Tree-basedORAM.Access(op, id, data*):
1: $x \longleftarrow$ pos[id]
2: pos[id] $\overset{\$}{\longleftarrow} [0 \ldots 2^{L-1}]$
3: $S \longleftarrow$ ReadPath($x$)
4: data $\longleftarrow$ Read block $a$ from $S$
5: if op = write then
6:     $S \longleftarrow (S - \{(\mathsf{id}, \mathsf{data})\}) \cup \{(\mathsf{id}, \mathsf{data}^*)\}$
7: $S \longleftarrow$ Evict($x'$), where $x'$ is the selected eviction path
8: return data
```

Figure 4. General access procedure in tree-based ORAM schemes.

An ORAM scheme is secure *if for any two data request sequences $\vec{o}$ and $\vec{o}'$ of the same length, their access patterns $\boldsymbol{AP}_j(\vec{o})$ and $\boldsymbol{AP}_j(\vec{o}')$ are perfectly/statistically/computationally indistinguishable.*

Tree-based ORAM. Most efficient ORAM schemes follow tree paradigm first proposed by Shi et al. [62]. Three components construct tree-based ORAM schemes: (*i*) a complete binary tree with N leaf nodes that can store up to N data blocks; (*ii*) a position map (pos) that associates each data block with a random leaf node (i.e., path ID); (*iii*) a stash component (S) to temporarily store some blocks that cannot be pushed into the tree during an ORAM access.

Each node in the tree is called a *bucket*, which can store up to Z blocks. Because the tree accommodates more slots (i.e., $Z \cdot N$) than the number of real blocks (N), tree-based ORAM fills empty slots with dummy blocks. All real and dummy blocks are IND-CPA encrypted to be indistinguishable. While the untrusted server stores these encrypted tree structures, the client maintains the position map and the stash S.

Each access in a tree-based ORAM incurs a read and an eviction operation, and Figure 4 describes the access in detail. Given an identifier id of a block that is being accessed, the client retrieves its location in the tree from the position map, and then requests the path from server (ReadPath). Retrieved blocks are temporarily stored in the stash S, and then the blocks are pushed back to the tree via an eviction operation on a selected path (Evict). Note that once the block is evicted, it must reside somewhere in the intersection nodes between the eviction path and its assigned path. Moreover, once the block is accessed, its location is updated with a new path ID selected uniformly at random.

Path-ORAM. Path-ORAM [67] has simple read and eviction procedures. The client reads all data in the path, discards dummy blocks and then, puts all real blocks into the stash. The eviction path is the same as read path. Blocks in the stash are pushed back into the empty slots on the eviction path, starting from the leaf bucket to the root bucket. The stash size in Path-ORAM was proven to be upper-bounded by the security parameter λ, which characterizes the overflow probability and statistical security, i.e., $|S| = O(\lambda)$ blocks. We describe the detailed algorithms of Path-ORAM in Appendix.

Circuit-ORAM. Circuit-ORAM [72] reduces the circuit complexity of Path-ORAM by minimizing the number of blocks to be involved during read and eviction. Each bucket has a meta-data that stores the information about real blocks and their path ID.

Read: Similar to the original tree ORAM in [62], where the client reads all data in the path, but only keeps the block of interest in the stash and removes it from the path. The removal process can be implemented efficiently by only flipping one bit in the bucket meta-data.

Eviction: The client prepares a list of blocks to be pushed down in the eviction path by scanning the meta-data of buckets in the path. Client picks one block in the stash (if any) that can be pushed to the deepest level of the tree, and then traverses from the root to leaf node. In each level, the client drops the on-hold block and picks at most one block to be put into a deeper level. For each data access, the client invokes two eviction procedures, with eviction path being selected randomly or deterministically as in [27] (see Appendix for details).

Circuit-ORAM incurs approximately 1.25× more I/O access than Path-ORAM. However, it has a smaller circuit size, and therefore, it is more efficient to be implemented with Intel SGX [61]. Circuit-ORAM has a smaller bucket size ($Z = 2$ vs. $Z = 4$ in Path-ORAM) and therefore, incurs less server storage. Similar to Path-ORAM, the stash size in Circuit-ORAM is shown to be a function of security parameter, i.e., $|S| = O(\lambda)$ blocks.

3.2 System Building Blocks

We use Intel SGX as a trusted execution environment to protect the execution of ORAM controller on the untrusted server. We run the logic in an SGX enclave, which guarantees the isolation and confidentiality of its execution, to protect the ORAM controller logic from attacks. We utilize the remote attestation protocol of Intel SGX to check the integrity of our logic and securely exchange/provision secret keys for storing ORAM data structures as well as protecting communication channels between the enclave and the client. We also implement our logic in the enclave using `CMOV` instructions to prevent potential access pattern leakage.

As building blocks of POSUP, we utilize three main components of **Intel SGX**:

• *Enclave*: An enclave is a trusted execution unit of Intel SGX and it is protected by isolation and integrity/confidentiality guarantees. Intel SGX isolates the enclave's execution by providing a private memory called the *enclave page cache* (EPC), which resides a reserved space in DRAM (the processor reserved memory, PRM) [17]. The EPC is isolated from the other software security domains by SGX's hardware mechanism, thus SGX blocks any software attempt to read/write enclave's memory from user-level as well as privileged-level (including operating systems and virtual machine monitor) attackers. Moreover, because SGX encrypts (with integrity check) the data before storing it on to DRAM, EPC only stores encrypted data on it, and thereby any hardware attempt to read enclave's memory will not leak any meaningful information, and any tampering to enclave's memory will be detected (and then SGX stops the enclave's execution).

• *Remote attestation and key exchange*: SGX supports the remote attestation of the enclave to authenticate if the configuration of the enclave is correct and to share a secret key for secure communication [39, 17] only after the authentication. When a remote attestation request initiated by a client is delivered to the enclave, the SGX subsystem will run the trusted quoting enclave, which creates a measurement (i.e., hash of configurations, loaded program with a nonce, and a public key material for key exchange) of the enclave and signs it (with the quoting enclave's key). This measurement will be submitted to the Intel Attestation Service (IAS) to verify the quoting enclave's signature, and the result will be signed by IAS, and then will be delivered to the client. By verifying IAS's signature on the result, the client ensures that a correct enclave is running on the server. The attestation message also includes public key parameters of Diffie-Hellman key exchange of the client and the enclave so that a client can securely communicate with the enclave after this process, by encrypting data using the shared secret.

• *System calls*: Since the enclave is a part of user-level process, Intel SGX does not provide any protection on privileged operations like system calls such as file and network I/O, etc. Because such operations have to be performed by the untrusted OS, the enclave must encrypt data before transferring them to the OS. For example, a network communication between the client and the enclave should apply encryption to their connection, and a file write operation should store only encrypted data. For this purpose, Intel provides cryptographic libraries and tools for secure data migration between the enclave

```
$B \leftarrow \mathsf{OGet}(S, \mathsf{bID})$:
1: $B \leftarrow \perp$
2: for $i = 1, \ldots, |S|$ do
3:     $v \leftarrow \mathsf{ocmp}(S[i].\mathsf{bID}, \mathsf{bID})$
4:     $B \leftarrow \mathsf{oupt}(v, S[i], B)$
5: return $B$
```

Figure 5. OGet function in POSUP.

and the OS, so the enclave can securely communicate across the security boundary if it is provisioned with a secret key for the encryption; and this can be done via remote attestation.

Secure Operations inside Enclave. We implement oblivious assignment (oupt) and oblivious equality comparison (ocmp) functions based on CMOV instructions proposed in [59, 54], which do not leak access pattern via control-flow side-channel when POSUP executes ORAM controller inside the enclave as follows.

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

We refer interested readers to [59, 54] for detailed description of these functions. Note that our ocmp function slightly differs from what being originally proposed in [54], where we employ SETE, instead of SETG instruction for equality checking.

ZeroTrace [61] proposed OReadPath and OEvict, which are secure versions of ReadPath and Evict tree-based ORAM functions, respectively, both of which are executed by the enclave without leaking side-channel access pattern. We implemented our version of OReadPath and OEvict and refer readers to ZeroTrace [61] for detailed description. In OReadPath and OEvict, we scan the entire stash and path, then use ocmp and oupt to put real blocks from the path to the stash, or vice versa. This results in Path-ORAM to be more costly than Circuit-ORAM when dealing with large dataset. Since Path-ORAM pushes all real blocks from the path to the stash or vice-versa, its OReadPath and OEvict incur two nested loops, where we must scan the entire stash for each path slot access to hide the access pattern. Since Circuit-ORAM only processes one targeted block at a time, it only incurs two separate loops, which scan the entire path once to get target blocks and then, the entire stash. This makes Circuit-ORAM more efficient than Path-ORAM when dealing with large dataset (see §6). We then implement OGet function (Figure 5), which reads a block ID from the stash S into the enclave without leaking access pattern via ocmp and oupt.

4 The Proposed Platform

We first describe the oblivious data structures in POSUP.

We then present the oblivious search and update protocol in details.

4.1 Oblivious Data Structures

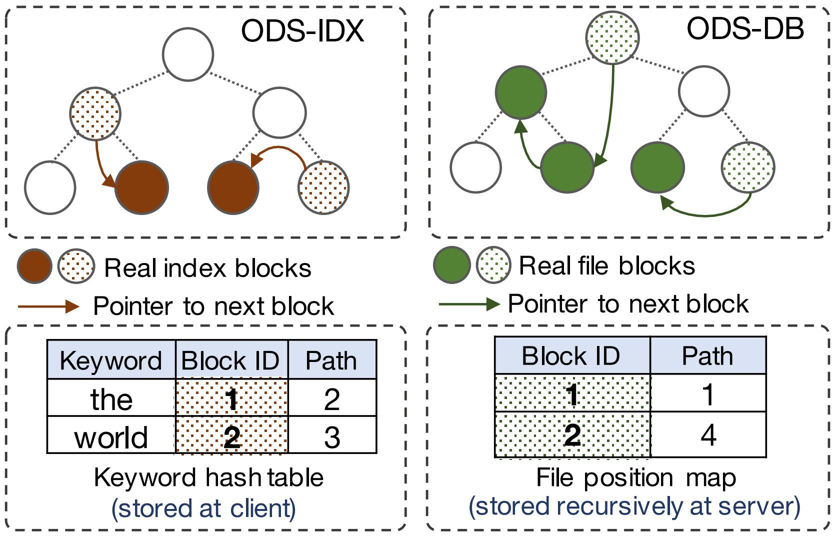
Figure 6 presents the overview of ODS-IDX and ODS-DB in POSUP. ODS-IDX and ODS-DB follow the tree ORAM paradigm presented in §3.1, because POSUP harnesses Path-ORAM and Circuit-ORAM as oblivious memory access primitives. We first create a search index IDX from a set of plaintext files DB, and then package IDX and DB into ODS-IDX, ODS-DB, respectively, as follows.

We construct IDX as an inverted index, in which given DB as the input, we extract unique keywords and associate each keyword w_i with the list of corresponding file identifiers id_{ij} that w_i appears in as $w_i := \left(\mathsf{id}_{i_1} \ldots, \mathsf{id}_{i_n}\right)$. We divide the list of each keyword in IDX into multiple chunks of the same size, and package them into separate tree ORAM blocks. We use the pointer trick (i.e., linked-list in [74]) to connect these blocks with each other, where the information of successive blocks are stored in their predecessors. Thus, each block is in form of $B := \left(\mathsf{bID}, \mathsf{DATA}, \mathsf{NextID}, \mathsf{NextPath}\right)$, where bID is the block ID, DATA contains the block data (e.g., partial list of file IDs), NextID and $\mathsf{NextPath}$ are the ID and the path of the next block, respectively. Finally, we create ODS-IDX by putting all constructed blocks into a tree ORAM structure.

We apply the same principle as above to construct ODS-DB from DB. Since each file is organized into a linked-list, we set $B.\mathsf{bID} = \mathsf{id}$, where B is the first block in the list and id is the identifier of the file that B represents. Given that the size of IDX is generally smaller than that of DB, we build ODS-IDX and ODS-DB as two separate oblivious data structures, where the block size of ODS-IDX is smaller than that of ODS-DB.

POSUP requires to maintain the file position map to keep track of the path of the starting blocks in each ODS linked-list. Note that we can index file identifiers with an integer from 1 to N, where N is the total number of files in DB, since POSUP focuses on search and update functionalities on keywords appearing in DB. This allows us to maintain the file position map via a recursive ORAM in the server's side. Our design only needs to perform recursive ORAM access to get the path of starting block, while the path of other blocks in the linked-list is obtained in their predecessor due to ODS technique. This reduces the number of recursive calls on the file position map component, compared with if we directly apply oblivious memory primitives (e.g.,[23, 3]) to enable oblivious search functionality.

We present the detailed algorithm to construct ODS-IDX and ODS-DB in Figure 7. We encrypt ODS-IDX and ODS-DB with ORAM key k_o and store ODS-DB and ODS-IDX on the server's untrusted



Keyword	Block ID	Path
the	1	2
world	2	3

Block ID	Path
1	1
2	4

Figure 6. Illustration of ODS-IDX and ODS-DB packaged into ORAM tree in our platform. Blanked nodes denote a bucket containing all dummy blocks, while colored nodes denote a bucket containing at least one real file/index block.

```
ODS.Setup(DB):
1:  $\mathcal{B} \leftarrow \emptyset; \mathcal{B}' \leftarrow \emptyset$
2:  $\mathcal{W} := (w_1 \ldots, w_M) \leftarrow$ Extract all unique keywords in DB
3:  Construct inverted index IDX $\leftarrow \left( \langle w_i, \vec{\mathsf{id}}_i \rangle_{i=1}^{M} \right)$, where $\vec{\mathsf{id}}_i := (\mathsf{id}_{i_1}, \ldots, \mathsf{id}_{i_n})$ are file IDs $w_i$ appears in
4:  for each file $f_i \in \mathcal{F}$ do
5:      Split $f_i$ into $m_i$ chunks of size $|B|$
6:      $B_{ij}$.DATA $\leftarrow j$-th chunk; $B_{ij}$.pID $\xleftarrow{\$} [2^L]$ $\forall j \in [m]$
7:      for $j = 1, \ldots, m_i - 1$ do
8:          $B_{ij}$.NextID $\leftarrow B_{ij+1}$.bID
9:          $B_{ij}$.NextPath $\leftarrow B_{ij+1}$.pID
10:     $\mathsf{pos}_f[B_{i1}.\mathsf{bID}] \leftarrow B_{i1}$.pID
11:     $\mathcal{B} \leftarrow \mathcal{B} \cup \{B_{i1}, \ldots, B_{im_i}\}$
12: ODS-DB $\leftarrow$ BuildORAMTree$_{k_o}(\mathcal{B})$
13: BuildRecursiveORAM($\mathsf{pos}_f$)
14: for each keyword $w_i \in \mathcal{W}$ do
15:     $\vec{\mathsf{id}}_i \leftarrow$ IDX$[w_i]$; Split $\vec{\mathsf{id}}_i$ into $m'_i$ chunks of size $|B'|$
16:     $B'_{ij}$.DATA $\leftarrow j$-th chunk; $B'_{ij}$.pID $\xleftarrow{\$} [2^{L'}]$ $\forall j \in [m']$
17:     for $j = 1, \ldots, m'_i - 1$ do
18:         $B'_{ij}$.NextID $\leftarrow B'_{ij+1}$.bID
19:         $B'_j$.NextPath $\leftarrow B'_{ij+1}$.pID
20:     $\mathsf{pos}_w[B'_{i1}.\mathsf{bID}] \leftarrow B'_{i1}$.pID
21:     $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{B'_{i1}, \ldots, B'_{im'_i}\}$
22: ODS-IDX $\leftarrow$ BuildORAMTree$_{k_o}(\mathcal{B}')$
```

Figure 7. Setup algorithm to construct oblivious data structures in our system. 2-3: Construct inverted index IDX from files DB. 4-12: Build ODS-DB from DB. 13: Build recursive ORAM structure for file position map pos_f. 14-22: Build ODS-IDX from IDX. All data in ORAM structures (12,22) are encrypted with ORAM key k_o. B, B' denote ORAM blocks of ODS-DB and ODS-IDX resp. $|B|$ is the data size of block B. $[x]$ denotes $\{1, \ldots, x\}$.

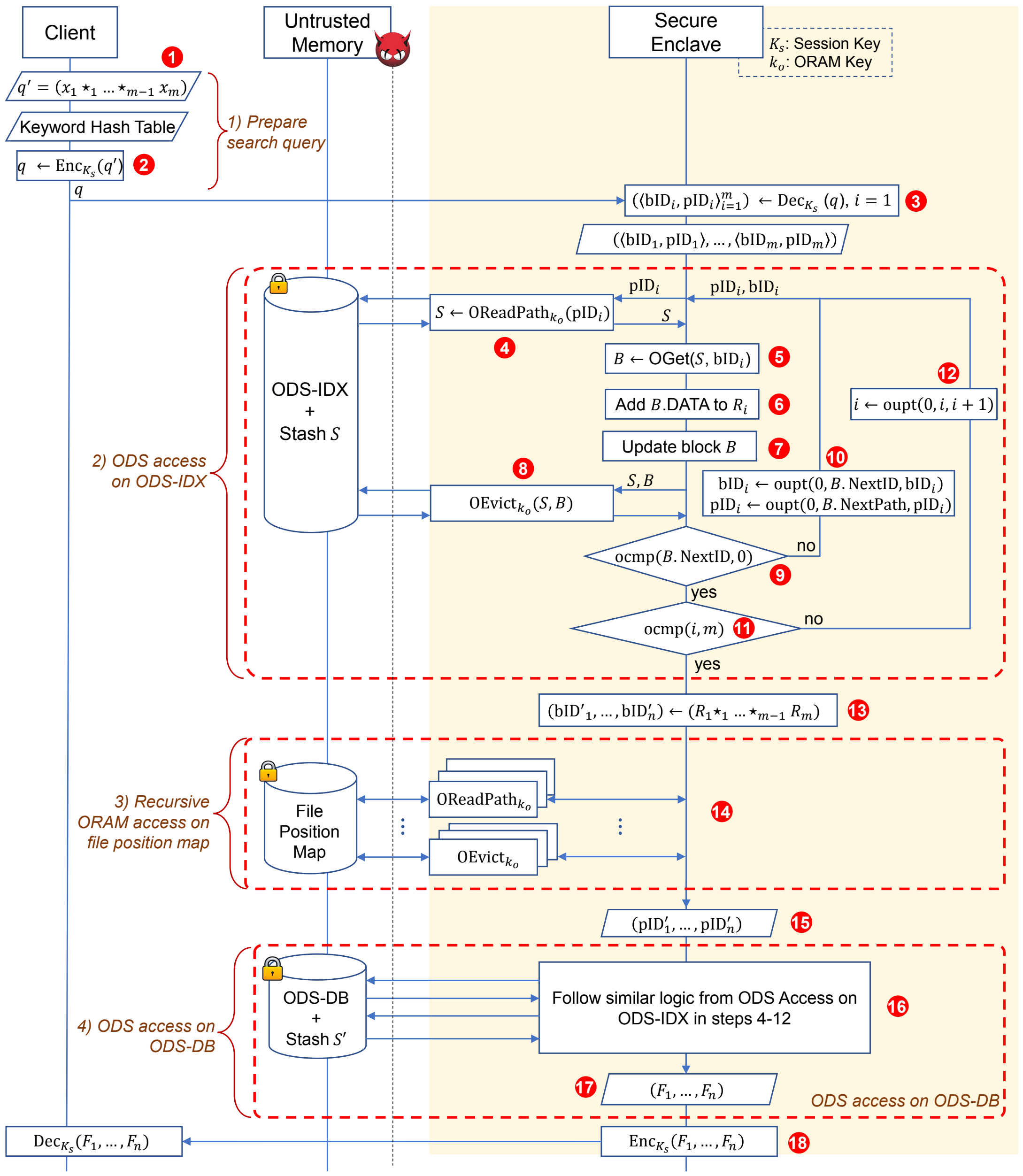
memory. The stash components required by ORAM operations on ODS-IDX, ODS-DB are also encrypted with k_o and stored in the server's untrusted memory. They are loaded and decrypted in the enclave when needed.

4.2 Search and Update Protocols

4.2.1 Oblivious Search

We describe our search protocol supporting boolean query in Figure 8 with the following outlines.
1) Prepare search query: Given a boolean query of the form $q' = w_1 \star_1 \cdots \star_{m-1} w_m$, where $\star_i \in \{\vee, \wedge\}$, the client generates a query of the form $q = \left(\langle \mathsf{bID}_1, \mathsf{pID}_i \rangle \star \cdots \star \langle \mathsf{bID}_m, \mathsf{pID}_m \rangle \right)$, where $\langle \mathsf{bID}_i, \mathsf{pID}_i \rangle \leftarrow \mathsf{pos}_w[w_i]$ are the first block ID and its path of the keyword w_i in ODS-IDX (❶), respectively. The client encrypts q with session key K_s and sends it to the server to be passed into the secure enclave (❷).
2) ODS access on ODS-IDX: The enclave decrypts q with K_s and initializes a memory R_i to temporarily store file IDs of each searched term w_i (❸). For each bID_i in q, it performs a sequence of oblivious accesses on ODS-IDX to retrieve all blocks in the linked-list, all of which form the entire list of file IDs

matching with w_i. Specifically, the enclave first executes OReadPath with path pID_{i1} (4) to fetch the block with bID_i from ODS-IDX into the stash S. It then reads the block from S via OGet (5) and then, stores the block data into R_i (6). Once a block bID_i is accessed, it will be assigned to a new random path according to the tree-based ORAM operation policy. Therefore, to achieve consistency, the enclave pre-



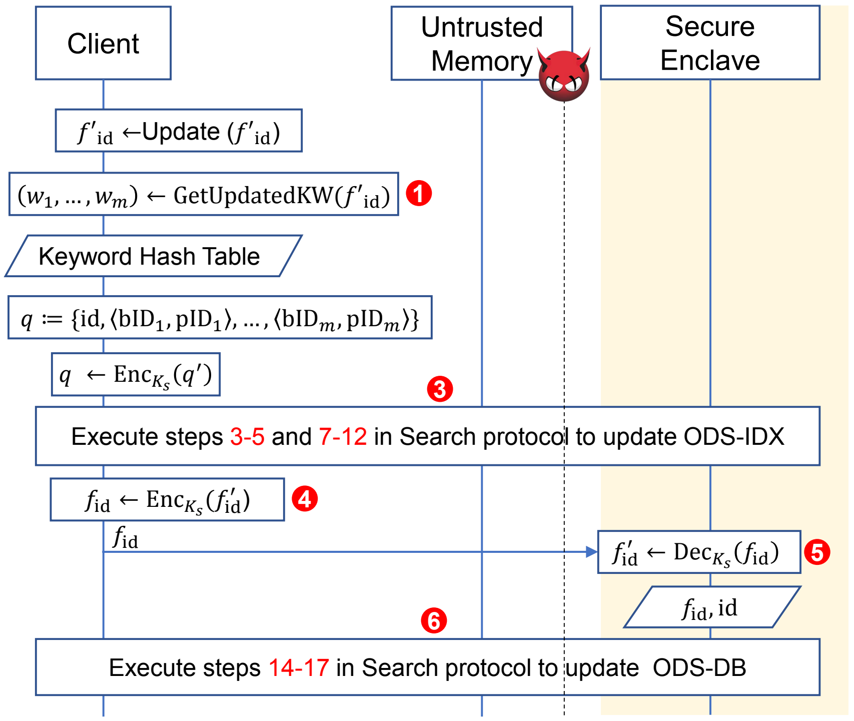
Client
Untrusted Memory
Secure Enclave
K_s: Session Key
k_o: ORAM Key
1
$q' = (x_1 \star_1 \dots \star_{m-1} x_m)$
Keyword Hash Table
$q \leftarrow \mathrm{Enc}_{K_s}(q')$
2
1) Prepare search query
q
$(\langle \mathrm{bID}_i, \mathrm{pID}_i \rangle_{i=1}^{m}) \leftarrow \mathrm{Dec}_{K_s}(q),\ i = 1$
3
$(\langle \mathrm{bID}_1, \mathrm{pID}_1 \rangle, \dots, \langle \mathrm{bID}_m, \mathrm{pID}_m \rangle)$
ODS-IDX + Stash S
pID_i
$\mathrm{pID}_i, \mathrm{bID}_i$
$S \leftarrow \mathrm{OReadPath}_{k_o}(\mathrm{pID}_i)$
S
4
$B \leftarrow \mathrm{OGet}(S, \mathrm{bID}_i)$
5
Add B.DATA to R_i
6
Update block B
7
12
$i \leftarrow \mathrm{oupt}(0, i, i+1)$
8
10
S, B
$\mathrm{OEvict}_{k_o}(S, B)$
$\mathrm{bID}_i \leftarrow \mathrm{oupt}(0, B.\mathrm{NextID}, \mathrm{bID}_i)$
$\mathrm{pID}_i \leftarrow \mathrm{oupt}(0, B.\mathrm{NextPath}, \mathrm{pID}_i)$
2) ODS access on ODS-IDX
$\mathrm{ocmp}(B.\mathrm{NextID}, 0)$
no
9
yes
$\mathrm{ocmp}(i, m)$
11
no
yes
$(\mathrm{bID'}_1, \dots, \mathrm{bID}'_n) \leftarrow (R_1 \star_1 \dots \star_{m-1} R_m)$
13
File Position Map
$\mathrm{OReadPath}_{k_o}$
OEvict_{k_o}
14
3) Recursive ORAM access on file position map
$(\mathrm{pID}'_1, \dots, \mathrm{pID}'_n)$
15
ODS-DB + Stash S'
Follow similar logic from ODS Access on ODS-IDX in steps 4-12
16
4) ODS access on ODS-DB
17
$(F_1, \dots, F_n)$
ODS access on ODS-DB
$\mathrm{Dec}_{K_s}(F_1, \dots, F_n)$
$\mathrm{Enc}_{K_s}(F_1, \dots, F_n)$
18

Figure 8. Detailed search protocol. 1-2 Client generates and sends encrypted search query q to the secure enclave. 3-12 Enclave decrypts q and performs ODS accesses on ODS-IDX to retrieve file IDs matching with each search term in q. 13-15 Enclave performs union/intersection over file ID list of search terms, and performs recursive ORAM to get the location (path) of file IDs on ODS-DB in the joint list. 16-18 Enclave performs ODS accesses on ODS-DB to retrieve all file contents, encrypts them and sends to client.

generates the random path pID' for the next block and updates this info in bID_i as $\mathsf{bID}_i.\mathsf{NextPath} = \mathsf{pID}$ (❼). Once the next block of bID_i is accessed, the enclave will use pID' as its new random path ID. The enclave performs OEvict to write the updated block back into ODS-IDX. Afterwards, it determines if there is a block linked with bID_i via ocmp (❾). If so, it gets the next block information from bID_i via ocmp and continues the above procedures until the final block in the linked-list is reached (❿). Otherwise, the enclave processes the next block bID_{i+1} in q by repeating the above procedures until the final block is reached (⓫-⓬).
3) Recursive ORAM access on file position map: After file IDs (stored in R_i) of each keyword w_i are retrieved, the enclave performs union/intersection on R_i according to $\star_i$ to get the final list of file IDs matched with q (⓭). For each block bID'_i in the joint list, the enclave performs recursive ORAM access (⓮) on the file position map structure to retrieve the corresponding path pID'_i of bID' in ODS-DB (⓯).
•*ODS Access on ODS-DB*: The enclave performs a sequence of ODS accesses on ODS-DB as similar to ODS-IDX accesses to retrieve the file content of for each bID'_i (⓰-⓱). Finally, the enclave encrypts all the retrieved files with the session key K_s, and sends them to the client (⓲).

4.2.2 Oblivious Update

Figure 9 outlines the oblivious update protocol. For simplicity, we assume that the file to be updated is already at the client's side. If it does not exist, it is straightforward to retrieve it via the oblivious search protocol (steps ⓮-⓱ in Figure 8). The main steps are as follows.



Client
Untrusted Memory
Secure Enclave
$f'_{\mathrm{id}} \leftarrow$ Update (f'_{id})
$(w_1, \ldots, w_m) \leftarrow \mathrm{GetUpdatedKW}(f'_{\mathrm{id}})$ ❶
Keyword Hash Table
$q := \{\mathrm{id}, \langle \mathrm{bID}_1, \mathrm{pID}_1\rangle, \ldots, \langle \mathrm{bID}_m, \mathrm{pID}_m\rangle\}$
$q \leftarrow \mathrm{Enc}_{K_s}(q')$
❸
Execute steps 3-5 and 7-12 in Search protocol to update ODS-IDX
$f_{\mathrm{id}} \leftarrow \mathrm{Enc}_{K_s}(f'_{\mathrm{id}})$ ❹
f_{id}
$f'_{\mathrm{id}} \leftarrow \mathrm{Dec}_{K_s}(f_{\mathrm{id}})$ ❺
$f_{\mathrm{id}}, \mathrm{id}$
❻
Execute steps 14-17 in Search protocol to update ODS-DB

Figure 9. Our oblivious update protocol. ❶-❸: Update ODS-IDX. ❹-❻: Update ODS-DB.

1) Update ODS-IDX: The client updates the file content and forms a list of keywords to be added or deleted in the updated file (❶). The client creates a query containing the updated file id along with (block ID-path) pairs of updated keywords. The client encrypts q with K_s and sends it to the enclave

(❷). The enclave decrypts q with K_s and then executes ODS access on ODS-IDX as presented in Search protocol to retrieve blocks indicated in q (❸). Once the block is read, the enclave updates block data by deleting or adding id.

2) Update ODS-DB: The client encrypts the updated file with K_s and sends it to the server to be passed into the enclave (❹). The enclave executes ORAM access on file position map to retrieve the path of updated file in ODS-DB, and then executes ODS access on ODS-DB to update file blocks, as in Search protocol (❻).

Optimization to ODS to improve update efficiency. Let m be the number of updated keywords w_i in the updated file f_{id}. The presented update protocol performs $\sum_{i=1}^{m}\lceil\frac{|w_i|}{|B|}\rceil$ ORAM accesses on ODS-IDX, where $|B|$ and $|w_i|$ are the block size and the number of linked-list blocks of w_i, resp., to find and add/delete id in the index block data. The cost might be high if w_i are most common keywords. We propose an optimization method on ODS to reduce the update cost from $\sum_{i=1}^{m}\lceil\frac{|w_i|}{|B|}\rceil$ to m as follows.

Intuitively, for each keyword w_i to be updated (add or delete) in f_{id}, the client requests the enclave to add id with a (add/delete) flag into an empty data slot in the first block of current linked-list of w_i. If there is no available slot, the enclave adds id and the flag into an empty block in ODS-IDX and links it with the current linked-list of w_i. This strategy results in first blocks in the list always containing the latest updated file IDs. Once a keyword w_i is searched, the enclave will clear stale IDs appearing in subsequent blocks in the list when accessing ODS-IDX.

4.3 Security

ODS and ORAM. We design and build POSUP by using ODS and ORAM, and therefore, its security is inherited from the security of these tools. Specifically, these tools guarantee that POSUP hides all access patterns on ODS-IDX and ODS-DB, given that they have the same length as shown in Definition 1. Therefore, any search/update operations involved with the *same* number of recursive ORAM and ODS accesses (e.g., search queries result in the same number of files to return) are *indistinguishable*. However, ORAM and linked-list ODS cannot hide the number of ORAM accesses. Therefore, POSUP leaks size access pattern as in its underlying primitives. This problem can be mitigated by adding padding; for instance, for the query that requires less than n total ORAM accesses, one can add dummy random ORAM access on ODS-IDX and ODS-DB, and dummy recursive ORAM access on file position map to quantize the their total number to be n thus *indistinguishable* by the attacker. Such padding strategies are generally dataset- and application-dependent, and also might incur heavy bandwidth and processing blowup. We refer readers to Ryoan [36], for its parts of data oblivious communication as well as quantizing processing time to learn more on how the quantization by padding can thwart such side-channel.

Side-channel attacks against Enclave. While the enclave of Intel SGX provides security guarantees of data confidentiality and integrity against direct memory access attacks, it is not free from side-channel attacks. POSUP does not aim to defeat all of the side-channel attacks, which is an impossible goal; instead, we try to build POSUP as a best-effort approach to defeat known side-channel attacks. Use of recursive ORAM and ODS in POSUP naturally defeats side-channel attacks that target data access patterns such as cache side-channel [32, 10, 30]. Employing oblivious data comparison (ocmp) and oblivious data assignment (oupt) in POSUP (see §3.2) defeats attacks that target data from the control-flow side-channel [47, 76, 11] because these primitives eliminates conditional branches on processing secrets, therefore such attacks cannot measure a difference in control-flow for different secrets.

5 Implementation

We implemented POSUP with C/C++ using the Intel SGX SDK v1.7. Our implementation contains a total of around 4.9K lines of code for trusted and untrusted modules as shown in Table 1. For cryptographic operation inside the enclave, we leveraged Intel SGX SDK library functions including `sgx_aes_ctr_encrypt` for encrypting ORAM with AES-CTR mode and `sgx_read_rand` for pseudo-random number generation. We implemented Path-ORAM and Circuit-ORAM controllers in the enclave to execute ODS access on ODS-DB and ODS-IDX. As mentioned in §4, our platform stores the ORAM stash components in the untrusted memory (RAM/SSD, encrypted), and the entire stash is loaded into the enclave when needed.

Table 1. Lines of code for each module in POSUP. The untrusted modules contain functionalities for (i) setting up ODS-IDX and ODS-DB; (ii) I/O access to load ODS structures; (iii) Caching strategy in [49]. The enclave module contains Path-ORAM and Circuit-ORAM controllers interacting with the untrusted modules.

Component	Lines of code
Untrusted Module	3,637
Enclave Module	1,276

6 Evaluation

In this section, we first describe our configuration and evaluation methodology, and then present our main experimental results. Specifically, to evaluate POSUP, this section gives the answer to following questions:

1) How efficient it is to execute one recursive-ORAM/ODS access on encrypted data in POSUP (in §6.2.1)?

2) How efficient it is to perform a keyword search in POSUP, compared with existing solutions? What are the end-to-end delay and bandwidth/processing blowup (in §6.2.2)?

3) How efficient it is to perform an update in POSUP, compared with other techniques (in §6.2.3)?

6.1 Configuration and Methodology

Hardware. We evaluated the performance of our system on a commodity HP Desktop, which supports Intel SGX and is equipped with Intel E3-1230 v5 @ 3.4 GHz CPU, 16 GB RAM and 512 GB SSD.

Dataset. Our dataset is the full Wikipedia English corpus `enwiki v.20180120`. To extract text data from the corpus, we used `WikiExtractor` [5] `Python` script and extracted 5,554,594 distinct text-only articles (i.e., files in our term) from `enwiki`. To collect the keywords for search, we implemented a standard tokenization method to extract unique alphabetical and non-alphabetical keywords from the dataset. The total number of unique keywords in the dataset is 7,075,917, and the total number of keyword-file pairs is 863,782,383. The total size of database (DB) is 27 GB (on the disk), and the total size of index (IDX) is 6.9 GB. Figure 10 presents the size distribution of text articles in our dataset.

Network. To assume a general use case of a mobile client and a cloud server, we use Wi-Fi as the communication channel between the client and the server, and mimic the bandwidth and latency of

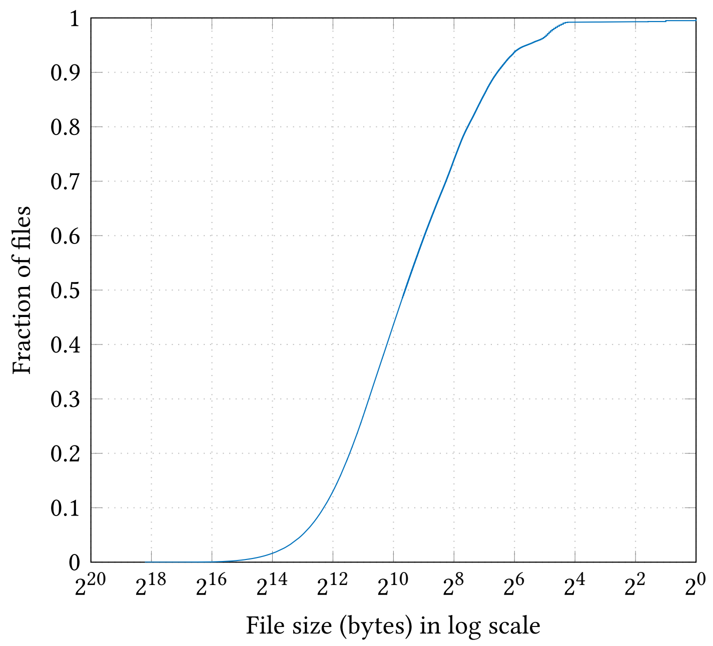


Figure 10. File size distribution in `enwiki` dataset in CDF. An (x, y) point denotes that y fraction of files are sized larger than x bytes. Based on this distribution, we choose the block size of ODS-DB as 3 KB because more than 50% of files are smaller than 3 KB.

using Amazon EC2 from our lab. The average network latency and transmission throughput are 18 ms and 150 Mbps, respectively.

Configurations and Methodology. We will compare POSUP to the implementation of existing designs, namely ORAM-SE and EntireSGX. The following represents the configuration of each implementation and how we compare each with POSUP.

• *POSUP configuration* We constructed ODS-IDX and ODS-DB with ORAM tree structures with 24 and 23 levels, respectively, to store the entire files (7,075,917, $\leq 2^{23}$) on them. Because more than 50% of files in our dataset are smaller than 3 KB (shown in Figure 10), we selected the block size of ODS-DB to be 3 KB to balance the efficiency and storage overhead. Likewise, we selected the block size of ODS-IDX to be 512 B, because the majority of search keywords is appearing in less than 512 files (see Figure 11c). For the stash size, we set the stash size of both ORAM schemes as 80 to achieve negligible overflow probability as presented in [67, 72].

• *ORAM-ODS-SE - Direct ORAM-SE composition in a traditional client-server model (without secure hardware).* To apply a fair comparison with POSUP, we use ODS in the ORAM-SE composition to apply the same optimization. We also set the size of ODS-IDX and ODS-DB as identical to POSUP. For the ORAM scheme, we employed Path-ORAM for ORAM-ODS-SE because it requires less access to ORAM, so it is more efficient than Circuit-ORAM in a standard client-server network setting. For the evaluation, we measured all delays when a client is accessing ODS and recursive ORAM on ODS-IDX and ODS-DB stored on the Amazon EC2 with the above network throughput and latency (18 ms and 150 Mbps). Note that our analysis is *conservative*, because we tend not to take into account the impact of side factors (e.g., disk I/O).

• *EntireSGX - Processing the entire outsourced data in Intel SGX.* This is an implementation that the entire ORAM reside in the enclave, and therefore, protected by Intel SGX. We measured the search delay by decrypting the entire EIDX and EDB inside the enclave. We used the maximum heap size (i.e., 95 MB) allowed to the enclave to subsequently decrypt EIDX and EDB to maximize its performance. For the update cost, we measured the delay of decryption and re-encryption of the entire EIDX and EDB inside the enclave. Because the size of our ODS-IDX and ODS-DB can allow to later add the same amount of dataset size in the setup phase (i.e., 27 GB file with 6.9 GB index), we double the size of EIDX and EDB in EntireSGX for fair comparison between two techniques.

6.2 Experiment Results

6.2.1 Micro Benchmark

POSUP is efficient, where it takes less than 1ms to access a 3 KB block in Circuit-ORAM sized 107 GB.

We first conducted a micro benchmark of POSUP to investigate the delay of performing a single recursive ORAM and ODS access. There are three factors that causes delay in each operation: (i) the time to read/write ORAM data from the hard disk to the memory and vice versa (i.e., I/O access); (ii) the time for the enclave to secure ORAM operations, such as applying encryption and decryption on the data (i.e., encryption overhead); (iii) the amount of data to be processed in each ODS operation on ODS-IDX and ODS-DB, expressed as::

$$D_{\mathsf{ODS}} = H \cdot |B| \cdot Z \cdot k, \tag{1}$$

where H and $|B|$ are the height and block size of ODS-IDX (or ODS-DB), respectively; and $(Z, k) = (4, 2)$ are the bucket size and the number of read/write operations in Path-ORAM, respectively. When Circuit-ORAM is used, $(Z, k) = (2, 5)$. The amount of data (in bytes) to be processed for each recursive ORAM on the file position map pos_f is:

$$D_{\mathsf{pos}_f} = \sum_{i=1}^{H_0} (H_{i-1} \cdot \; H_i \cdot Z \cdot k) \tag{2}$$

where $H_i = H_{i-1} - 1$ and H_0 is the height of ODS-DB.

Table 2 presents the execution time of each ODS access on ODS-IDX and ODS-DB and recursive ORAM on pos_f in our current configuration. Note that the performance changes for the different parameter configurations (e.g., for a diffent H, $|B|$, Z, or k) according to Equation 1 and Equation 2.

I/O Access. The I/O access in POSUP is efficient because we implemented caching techniques proposed in [49], where we cache the first K levels of the ORAM tree structures on the RAM. In this experiment, we used 4 GB of memory to cache 2/3 levels of both ODS-IDX and ODS-DB, which significantly reduced I/O delay from 520-767μs to ≤285μs. Compared to Path-ORAM, Circuit-ORAM incurs 1.25× more I/O access, and therefore its I/O latency is slightly higher than that of Path-ORAM. Because the recursive ORAM structure of file position map is small (i.e., ≈1 GB), we store the entire map directly on the RAM. This results in its I/O access delay to be negligible (i.e., ≤41 μs).

Enclave Process. Processing data in an enclave affects more on the access delay than I/O access because it handles encryption and decryption when reading data from ORAM. Another point that we observed from Table 2 is that the cost of executing Path-ORAM controller in the enclave is much higher than that of Circuit-ORAM because it has a larger circuit complexity. Specifically, when using Path-ORAM controller, the enclave must perform $O(\log N)\cdot|S|$ number of encryption/decryption, where $|S| = 80$ is the stash size. In contrast, using Circuit-ORAM controller requires $O(\log N) + |S|$ number of encryption/decryption. Therefore, our benchmarked result have shown that integrating Path-ORAM

Table 2. Execution time of a single ODS access on ODS-IDX and ODS-DB, and recursive ORAM accesses on file position map in POSUP. We ran each operation 500 times and took the average case number on the table.

Operation	**Execution Time(μs)**	
	Path-ORAM	**Circuit-ORAM**
ODS access on ODS-IDX		
I/O Access	**134**	144
Enclave Process	2,362	**686**
Total	2,496	**830**
ODS access on ODS-DB		
I/O Access	**156**	285
Enclave Process	3,909	**746**
Total	4,065	**1,031**
Recursive ORAM on file position map		
I/O Access	**34**	41
Enclave Process	13,246	**4,631**
Total	13,280	4,672

with secure hardware is much less efficient than Circuit-ORAM due to the multiplied factor $|S|$, which is 80. The process delay of recursive ORAM is high because this requires the enclave to perform additional ORAM encryption and decryption on $\log_2 N$ recursion levels.

Table 2 also illustrates that it takes 830 μs to obliviously access a 512 B block in ODS-IDX with Circuit-ORAM. That is, the latencies of performing single-keyword searches and boolean queries on ODS-IDX in many cases are likely to be similar to each other, and both of them are mostly dominated by the number of files to be returned (see §6.2.2).

We now illustrate the formula for calculating the number of ODS and recursive ORAM accesses incurred in each search and update query. Given a search query q with n keywords w_i, let m_i and q be the number of files matching with w_i and the final q, respectively. The search q on POSUP incurs $\sum_{i=1}^{n} \lceil \frac{m_i}{|B|} \rceil$ accesses on ODS-IDX plus m' recursive ORAM accesses on pos_f and plus $\sum_{i=1}^{m'} \lceil \frac{|f_i|}{|B|} \rceil$ accesses on ODS-DB, where B, B' are block sizes of ODS-IDX and ODS-DB, respectively, and $|f_i|$ is the size of file f_i in m' files. Given an updated file f with m updated keywords in it, the cost is $\sum_{i=1}^{n} \lceil m \rceil$ accesses on ODS-IDX plus 1 recursive ORAM accesses on pos_f plus $\lceil \frac{|f|}{|B|} \rceil$ accesses on ODS-DB.

Because the delay in I/O access and encryption in the enclave is stable (i.e., does not change between accesses), our measurement of the actual search and update delay in POSUP respected the above formulas and the micro benchmark in Table 2. In the following, we present actual benchmarked delay for search and update operations to showcase the efficiency of our system, compared with other techniques.

6.2.2 Search Delay

POSUP consumes $100\times$ less network bandwidth and requires $1,000\times$ less communication round-trips than ORAM-ODS-SE. POSUP incurs $10\times$ - $650\times$ less processing time in the enclave than EntireSGX for returning $\leq 2^{12}$ files as the search result, which falls in $> 99.5\%$ fraction of keywords that can be searched in our dataset. This results in the search delay of POSUP to be up-to $70\times$ and $400\times$ lower than ORAM-ODS-SE and EntireSGX, respectively.

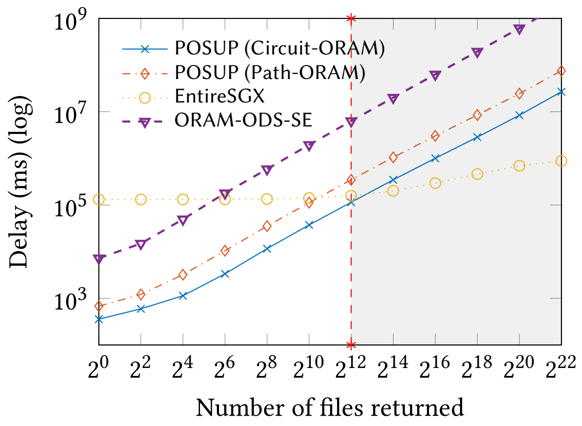
Figure 11a presents the end-to-end delay of processing a keyword search query in POSUP, compared with ORAM-ODS-SE and EntireSGX techniques. POSUP (the blue line) is *hundreds times faster* than ORAM-ODS-SE (the purple line) for any search query being performed. This is mainly because POSUP performs ORAM controlling operations in the enclave, so it does not incur significant network communication overhead like ORAM-ODS-SE as shown in Figure 11b; instead, the enclave reads a large amount of data from the memory, which is faster and cheaper than accessing over the network. ORAM-ODS-SE incurs overhead not only in bandwidth (i.e., $100\times$ more than POSUP) but also in generating a large number of network round-trips (i.e., $1000\times$ more than POSUP) because it must send/receive the entire recursive ORAM and ODS operations to the client. This is the main bottleneck of ORAM-ODS-SE, given that the network latency is hard to improve in practice.

When compared with EntireSGX, POSUP is *one to two orders of magnitude* faster than EntireSGX for more than 99.5% number of number keywords that can be searched. In Figure 11a, when searching keywords that returns $\leq 2^{12}$ files, POSUP is efficient than EntireSGX. Figure 11c presents the (accumulative) keyword distribution on `enwiki`. The Zipf's law distribution [53] showed in Figure 11c indicates that the cases that return $\leq 2^{12}$ files are the majority (99.5%). Therefore, POSUP is efficient than EntireSGX for a large fraction of keywords in practice. This is because the enclave in POSUP only works with a small amount of data (i.e., each ORAM access), while EntireSGX works with the entire ORAM as its working set, as presented in Figure 11d. For a small fraction of keywords (less than 0.5%), the end-to-end delay of POSUP is slower than that of EntireSGX. This is because a large number of ORAM and ODS accesses on ODS-IDX and ODS-DB requires data processing in enclave more than processing the entire dataset. The cost of executing Path-ORAM and Circuit-ORAM in POSUP is $C \cdot r \cdot 8\log_2(N)$ and $C \cdot r \cdot 10\log_2 N$, respectively, where r is the number of file blocks matched with the search query, $N = 2^{23}$ is the total number of file blocks in ODS-DB, and C is a constant factor. This formula implies that if $r \geq \frac{N}{Ck\log_2 N}$, where $k \in \{8, 10\}$ then loading the entire database into the enclave is better than performing ORAM. Our benchmark result of search in Figure 11a respects this formula. Theoretically, POSUP should incur more memory access than accessing the entire memory when returning more than 2^{15} files. The graph shows that overhead starts to matter returning around 2^{13} files. Therefore, we can assume the constant C as 4, and then POSUP incurs processing more than $\frac{N}{Ck\log_2 N}$ file blocks in the enclave.

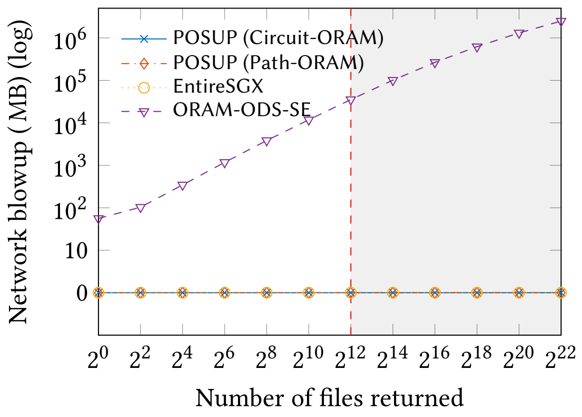
6.2.3 Update Delay

The update delay of POSUP is $40\times$ lower than ORAM-ODS-SE. This is because of the network bandwidth and round-trip overhead of ORAM-ODS-SE, as discussed in §6.2.2.

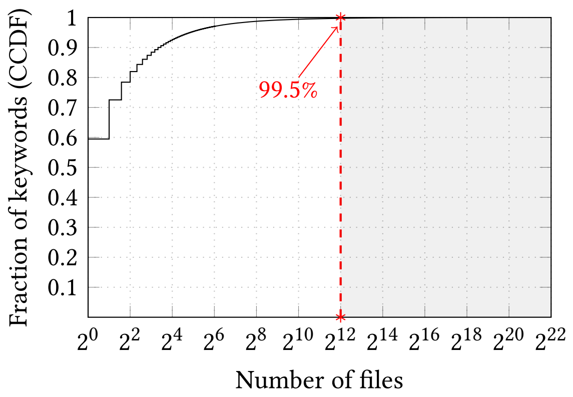
To measure the worst case performance, we selected the file with largest size (i.e., 290 KB) in `enwiki` and performed the update benchmark on that file for a different number of unique keywords that can be updated (add/delete) in it. Figure 12 presents the end-to-end update delay of POSUP and its counterparts.



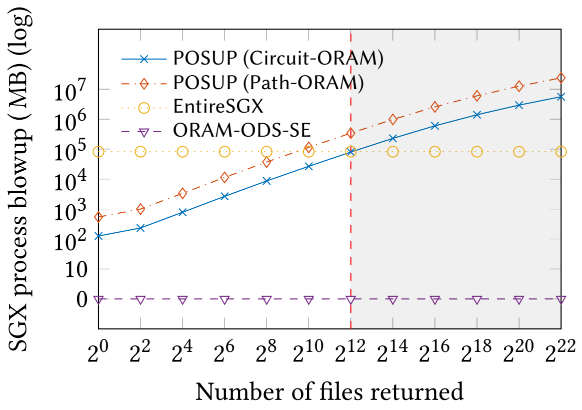
(a) End-to-end delay in POSUP and its counterparts regarding how many files returned in any single-keyword/boolean search.



(b) Network bandwidth blowup of POSUP and its counterparts. All hardware-assisted techniques do not incur network blowups.



(c) Keyword distribution in `enwiki` dataset. An (x, y) point denotes that y fraction of keywords appear in less than x files.



(d) SGX process blowup of POSUP and its counterparts. ORAM-ODS-SE technique does not require SGX process.

Figure 11. Detailed search delay of POSUP and its counterparts. In **(a)**, the delay of POSUP and EntireSGX was included with the time to transmit files to the client with 150 Mbps network throughput and 18ms latency. POSUP is more efficient than both EntireSGX and ORAM-ODS-SE for 99.5% fraction of keywords as indicated by the red dashed lines.

POSUP is one order of magnitude faster (40×) than ORAM-ODS-SE because it does not incur bandwidth and round-trip blowups as analyzed in §6.2.2. POSUP with Circuit-ORAM produces the highest throughput so that it achieves the lowest update delay among its counterparts.

POSUP is 5600× faster than EntireSGX, however, we would like to exclude this from the performance comparison because this is due to the inevitable overhead in disk writing required by the design of EntireSGX in an update operation. An update in EntireSGX requires re-encrypting the entire data and write them back to the disk. Therefore, the update delay of POSUP is *three orders of magnitude* faster than EntireSGX.

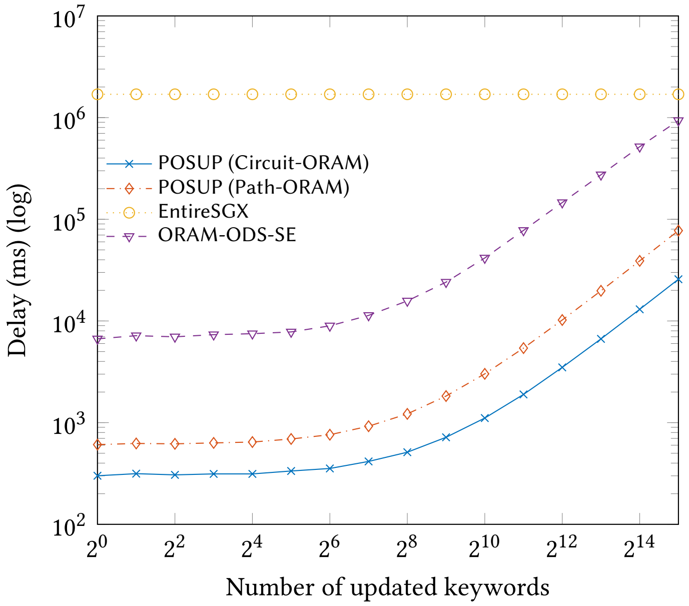


Figure 12. End-to-end delay of updating a 290 KB file with different number of updated keywords involved in.

6.2.4 Storage Overhead

The server storage overhead of EntireSGX is more efficient than that of POSUP and ORAM-ODS-SE. This is because in POSUP and ORAM-ODS-SE, IDX and EDB are arranged as tree-ORAM structures, which incur a constant (e.g., 1.5×-2×) size blowup. Specifically, the total server storage of POSUP is $|\mathsf{ODS\text{-}IDX}| + |\mathsf{ODS\text{-}DB}| + |\mathsf{pos}_f| = 70.1 + 103.8 + 1 = 174.9$ GB if POSUP uses Circuit-ORAM. The corresponding overhead is $140 + 207 + 1.5 = 348$ GB if POSUP uses Path-ORAM. Note that some capacity of ORAM structure is reserved to enable oblivious update (e.g., addition/deletion). Therefore, our server storage overhead presented above can allow to add 3× amount of IDX and DB presented in §6.1.

7 Related Work

Querying encrypted data. Searchable Encryption (SE) [63] enables the client to conduct search operations on encrypted data. Curtmola et al. [18] proposed a secure Symmetric SE (SSE) scheme that supports single-keyword search, followed by refinements with improved search functionalities (e.g., [12, 68, 71]) and security (e.g., [16, 44]). Kamara et al. [42] proposed a Dynamic SSE (DSSE) scheme that supports update functionality. Afterwards, several DSSE schemes proposed offering different features in terms of security (e.g., [9]), efficiency (e.g., [14, 43, 21]) and query functionalities (e.g., [15, 41, 70]). Another line of research focuses on developing encrypted query techniques that are compliant to legacy systems. For instance, ShadowCrypt [33] and Mimesis Aegis [45] propose encrypted search/update operations as an intermediate cryptographic service layer, which allows the client to interact with online applications without modifying server infrastructure. CryptDB [56] and its variants (e.g., [4, 1, 2, 55, 57]) leverage property-preserving encryptions (e.g., [8, 6]) to perform encrypted structured queries (e.g., SQL) to legacy database management systems (e.g., MongoDB).

Security vulnerability of encrypted search solutions. All aforementioned techniques leak access pattern that results in various types of statistical inference attacks. For instance, by exploiting access pattern leakages in DSSE, it is possible to determine which keyword has been searched with a high probability (e.g., [40, 48, 77, 13]). The legacy-compatible encrypted search techniques leak substantial additional information beyond the access pattern because of property-preserving encryption techniques [51, 31, 13, 58].

Solutions to remedy security vulnerabilities. ORAM [28] can hide both read and write access patterns, and therefore, it has been considered to seal such leakages in oblivious storage [66, 7, 65] and searchable encryption [25, 50]. Despite its merits, ORAM incurs a poly-logarithmic bandwidth blowup [67, 72], which has been shown to be extremely costly to be used for searchable encryption in the standard client-server network setting [50, 64, 7]. Although ORAMs with constant client-server bandwidth blowups have been proposed recently (e.g., [19, 34]), they either incur significantly higher delay than ORAMs with logarithmic-bandwidth blowup because of homomorphic encryption (e.g., [26]) or requiring multiple computing servers that which increase the deployment cost in practice.

The ORAM communication lower bound has been well-established [28, 72]. Thus, recent studies start to look for the support of secure hardware to make ORAM for client-server applications more practical. The idea of ORAM and secure-hardware composition was first suggested by concurrent studies in [49, 23, 60]. With the advent of widely available trusted execution environments on commodity hardware (e.g., Intel SGX), the deployment of hardware-supported cryptographic primitives has become more feasible. For instance, ZeroTrace [61] and Obliviate [3] leveraged Intel SGX with ORAM to enable oblivious memory primitives and file access operations, respectively. Intel SGX was also used to design a functional encryption framework in [22]. ObliDB [20] harnessed Intel SGX with Path-ORAM to enable oblivious SQL-queries on legacy database management systems.

8 Conclusion

In this paper, we developed a new SGX-supported oblivious search and update platform called POSUP. We achieved this by realizing highly optimized SGX-assisted oblivious data structures that enable practical oblivious search and update operations. We implemented and deployed POSUP on commodity hardware, and evaluated its performance on a very large dataset (full-size Wikipedia English corpus). Our experiments showed that POSUP achieves two to three orders of magnitude less bandwidth blowup and communication round-trips than traditional client-server model for oblivious search and updates. Similarly, POSUP offers one order of magnitude less processing blowup over alternatives that process entire data in SGX.

Our work confirms the extreme costs of ORAM(ODS)-SE composition in traditional client-server model, and demonstrates that hardware-assisted oblivious search and update systems as shown in POSUP are highly promising alternatives towards enabling privacy-preserving cyber-infrastructures.

References

[1] Always encrypted. https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine/.

[2] Google encrypted big query. https://github.com/google/encrypted-bigquery-client/.

[3] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious file system for intel sgx. 2018.

[4] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*. Citeseer, 2013.

[5] attardi. WikiExtractor. https://github.com/attardi/wikiextractor.

[6] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*, pages 535–552. Springer, 2007.

[7] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. 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.

[8] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 224–241. Springer, 2009.

[9] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. Technical report, IACR Cryptology ePrint Archive 2017, 2017.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, Aug. 2017.

[11] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[12] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems*, 25(1):222–233, 2014.

[13] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM CCS*, pages 668–679. ACM, 2015.

[14] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive*, 2014:853, 2014.

[15] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*, pages 353–373. Springer, 2013.

[16] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594, 2010.

[17] V. Costan and S. Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.pdf.

[18] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM CCS*, pages 79–88. ACM, 2006.

[19] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[20] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[21] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans. Efficient dynamic searchable encryption with forward privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(1):5–20, 2018.

[22] B. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov. Iron: functional encryption using intel sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 765–782. ACM, 2017.

[23] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

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

[25] S. Garg, P. Mohassel, and C. Papamanthou. Tworam: Round-optimal oblivious ram with applications to searchable encryption. *IACR Cryptology ePrint Archive*, 2015:1010, 2015.

[26] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

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

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

[29] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[30] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, 2017.

[31] P. Grubbs, T. Ristenpart, and V. Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 162–168. ACM, 2017.

[32] M. Hähnel, W. Cui, and M. Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[33] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1028–1039. ACM, 2014.

[34] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen. 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*, pages 491–505. ACM, 2017.

[35] T. Hoang, A. Yavuz, and J. Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016.

[36] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

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

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

[39] Intel Corporation. Intel Software Guard Extensions SDK for Linux OS (Developer Reference), 2016. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.7_Open_Source.pdf.

[40] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12, 2012.

[41] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 94–124. Springer, 2017.

[42] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 965–976. ACM, 2012.

[43] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1449–1463. ACM, 2017.

[44] K. Kurosawa and Y. Ohtaki. UC-secure searchable symmetric encryption. In *Financial Cryptography and Data Security (FC)*, volume 7397 of *Lecture Notes in Computer Science*, pages 285–298. Springer Berlin Heidelberg, 2012.

[45] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis aegis: A mimicry privacy shield-a system's approach to data privacy on public cloud. In *USENIX Security Symposium*, pages 33–48, 2014.

[46] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[47] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[48] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[49] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[50] M. Naveed. The fallacy of composition of oblivious ram and searchable encryption. In *Cryptology ePrint Archive, Report 2015/668*, 2015.

[51] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[52] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *Security and Privacy (S&P), 2014 IEEE Symposium on*, pages 639–654. IEEE, 2014.

[53] M. E. Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[54] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, pages 619–636, 2016.

[55] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *OSDI*, pages 587–602, 2016.

[56] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[57] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *NSDI*, pages 157–172, 2014.

[58] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1341–1352. ACM, 2016.

[59] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, pages 431–446, 2015.

[60] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. *ACM SIGARCH Computer Architecture News*, 41(3):571–582, 2013.

[61] S. Sasy, S. Gorbunov, and C. Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[62] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.

[63] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[64] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Annual Network and Distributed System Security Symposium – NDSS*, volume 14, pages 23–26, 2014.

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

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

[67] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications security*, pages 299–310. ACM, 2013.

[68] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. In *ACM SIGSAC AsiaCCS*, pages 71–82. ACM, 2013.

[69] W. Sun, R. Zhang, W. Lou, and Y. T. Hou. Rearguard: Secure keyword search using trusted hardware. In *IEEE INFOCOM*, 2018.

[70] B. Wang, S. Yu, W. Lou, and Y. T. Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *INFOCOM, 2014 Proceedings IEEE*, pages 2112–2120. IEEE, 2014.

[71] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *IEEE 30th International Conference on Distributed Computing Systems*, pages 253–262. IEEE, 2010.

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

[73] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. Scoram: oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202. ACM, 2014.

[74] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.

[75] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)*, Crete, Greece, Sept. 2016.

[76] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[77] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, 2016.

Appendix

We give the detailed algorithm of Path-ORAM in Figure 13. In this algorithm, $P(x, l)$ denotes the bucket at level l in the path x. ReadBucket(x) denotes reading all blocks in bucket x and WriteBucket(x, S') denotes writing a block S' into an empty slot in bucket x.

```
PathORAM.ReadPath($x$):
1: for $l = 1, \ldots, L$ do
2:     $S \longleftarrow S \cup \mathsf{ReadBucket}(P(x, l))$
PathORAM.Evict:
1: Let $x$ be the path revealed in PathORAM.ReadPath procedure
2: for $l = L, \ldots, 1$ do
3:     $S' \longleftarrow \{(a', \mathsf{data}') \in S : P(x, l) = P(\mathsf{pos}[a'], l)\}$
4:     $S' \longleftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$.
5:     $S \longleftarrow S - S'$
6:     $\mathsf{WriteBucket}(P(x, l), S')$
```

Figure 13. Access procedure in Path-ORAM.

We give the detailed algorithm of Circuit-ORAM with the deterministic eviction strategy in Figure 14, which execute subroutines in Figure 15. ReadAndRemove(x, id) returns a block with id, which is read and deleted from bucket x.

```
CircuitORAM.ReadPath($x$):
1: for $l = 0, \ldots, L$ do
2:     if $(a', \mathsf{data}) \longleftarrow \mathsf{ReadAndRemove}(P(x, l), a) \neq \perp$ then
3:         $S \longleftarrow S \cup (a', \mathsf{data})$
CircuitORAM.Evict:
1: Let $t$ be a global timestamp initialized with 0
2: $x \longleftarrow$ order-reversal of base-2 digits of $(t \mod 2^L)$.
3: $t \longleftarrow t + 1$
4: Execute PrepareDeepest($x$) and PrepareTarget($x$) subroutines to pre-process arrays deepest and target.
5: hold $\longleftarrow \perp$; dest $\longleftarrow \perp$
6: for $i = 0, \ldots, L$ do
7:     towrite $:= \perp$
8:     if (hold $\neq \perp$) and ($i$ = dest) then
9:         towrite $\longleftarrow$ hold
10:        hold $\longleftarrow \perp$; dest $\longleftarrow \perp$
11:    if target[$i$] $\neq \perp$ then
12:        hold $\longleftarrow$ read and remove deepest block in $P(x, i)$
13:        dest $\longleftarrow$ target[$i$]
14:    Place towrite into $P(x, i)$ if towrite $\neq \perp$.
15: Repeat steps 2-14 one time
```

Figure 14. Access in Circuit-ORAM with deterministic eviction.

```
PrepareTarget($x$):
 1: dest $\longleftarrow \perp$; src $\longleftarrow \perp$, target $\longleftarrow (\perp, \dots, \perp)$
 2: for $i = L, \dots, 0$ do
 3:     if $i$ = src then
 4:         target[$i$] $\longleftarrow$ dest; dest $\longleftarrow \perp$; src $\longleftarrow \perp$
 5:     if (dest = $\perp$ and $P(x, i)$ has empty slot) or (target[$i$] $\neq \perp$) and (deepest[$i$] $\neq \perp$) then
 6:         src $\longleftarrow$ deepest[$i$]
 7:         dest $\longleftarrow i$
PrepareDeepest($x$):
 1: deepest $\longleftarrow (\perp, \dots, \perp)$ src $\longleftarrow \perp$; goal $\longleftarrow -1$
 2: if $|S| > 0$ then
 3:     src $\longleftarrow 0$
 4:     goal $\longleftarrow$ Deepest level that a block in $P(x, 0)$ can legally reside on path.
 5: for $i = 1, \dots, L$ do
 6:     if goal $\geq i$ then deepest[$i$] $\longleftarrow$ src
 7:     $\ell \longleftarrow$ Deepest level that a block in $P(x, i)$ can legally reside on path.
 8:     if $\ell$ > goal then
 9:         goal $\longleftarrow \ell$
10:         src $\longleftarrow i$
```

Figure 15. Subroutines in Circuit-ORAM.