sciendo

Proceedings on Privacy Enhancing Technologies ; 2019 (1):172–191

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

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

**Abstract:** The 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 leaks in access patterns. In response to such attacks, the community has proposed the Oblivious Random Access Machine (ORAM). However, due to the logarithmic communication overhead of ORAM, the composition of ORAM and SE is known to be costly in the conventional client-server model, which poses a critical barrier toward its practical adaptations.

In this paper, we propose a novel hardware-supported privacy-enhancing platform called *Practical Oblivious Search and Update Platform* (POSUP), which enables oblivious keyword search and update operations on large datasets with high efficiency. We harness Intel SGX to realize efficient oblivious data structures for oblivious search/update purposes. We implemented POSUP and evaluated its performance on a Wikipedia dataset containing $\geq 2^{29}$ keyword-file pairs. Our implementation is highly efficient, taking only 1 ms to access a 3 KB block with Circuit-ORAM. Our experiments have shown that POSUP offers up to 70× less end-to-end delay with 100× reduced network bandwidth consumption compared with the traditional ORAM-SE composition without secure hardware. POSUP is also at least 4.5× faster for up to 99.5% of keywords that can be searched compared with state-of-the-art Intel SGX-assisted search platforms.

**Keywords:** Secure Enclaves, Intel SGX, Oblivious Data Structures, Oblivious Search/Update.

DOI 10.2478/popets-2019-0010
Received 2018-05-31; revised 2018-09-15; accepted 2018-09-16.

***Corresponding Author: Thang Hoang:** EECS, Oregon State University, E-mail: hoangmin@oregonstate.edu.
**Muslum Ozgur Ozmen:** EECS, Oregon State University, E-mail: ozmenmu@oregonstate.edu.
**Yeongjin Jang:** EECS, Oregon State University, E-mail: yeongjin.jang@oregonstate.edu.
**Attila A. Yavuz:** CSE, University of South Florida, E-mail: attilaayavuz@usf.edu. Part of this work done while the author was at Oregon State University.

## 1 Introduction

The 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, British Airways) 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 also prevent the user from performing even simple queries (e.g., search or update) on encrypted data, thereby diminishing the data utilization.

Searchable Encryption (SE) techniques have 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), both of which are outsourced to the cloud. One area of SE research focuses on designing new SE schemes (e.g., [14, 43, 55, 68]) with provable security that offer various trade-offs in terms of security, functionality, and efficiency. The other area of research aims to enable encrypted queries that are compliant with legacy infrastructure such as database management systems (e.g., mySQL, mongoDB) [1, 2, 4, 33, 41, 47, 58–60]. Despite their merits, all these techniques leak access patterns, which results in statistical analysis attacks (e.g., [13, 13, 31, 40, 50, 54, 61, 82]).

The Oblivious Random Access Machine (ORAM) [29] can hide access pattern and, therefore, it can seal the leaks in SE. Unfortunately, because ORAM is bandwidth-heavy, using ORAM for SE is costly in the conventional client-server model. To reduce this communication overhead, recent studies have investigated the support of ORAM with secure hardware [3, 24, 51, 63, 65]. 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 a legacy architecture (e.g., [24, 51, 63]). With the advent of trusted execution environments on commodity hardware, the deployment of such hardware-supported cryptographic primitives becomes more feasible. Intel SGX [17, 37, 38]

has been explored to enable efficient cryptographic operations such as oblivious memory access primitives [3, 65] and functional encryption [23].

**Our Objective.** The goal of this paper is to take the ORAM supported by a commodity secure hardware to the next level, in which we develop oblivious data structures using Intel SGX to offer practical oblivious search/update operations on very large datasets. We implemented our techniques to demonstrate their efficiency compared with state-of-the-art approaches.

## 1.1 Motivation

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

**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 was shown to be inefficient [7, 35, 53]. Naveed et al. [53] 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 high delays for a large dataset with standard network settings. Our experiment in section 6 has shown that this approach incurs one to two orders of magnitude more client-server bandwidth overhead for both keyword search and update operations.

**State-of-the-art Hardware-Supported Oblivious Search Platforms.** Existing systems [25, 73] require secure hardware (e.g., Intel SGX) to process the entire outsourced data (e.g., encryption/decryption) for each search query to completely hide the access pattern. Unfortunately, this approach might also incur a high delay when dealing with a large amount of outsourced data since its cost grows *linearly* with the database size. Moreover, state-of-the-art solutions did not fully investigate the update capability, which seems an essential feature of data-outsourcing applications.

ZeroTrace [65] proposed efficient oblivious memory primitives by harnessing recursive Circuit-ORAM with Intel SGX. This design is efficient for generic oblivious access purposes, where it does not require storing the position map and uses Circuit-ORAM, which is more computation-efficient than Path-ORAM when harnessed with secure hardware. Since the authors focus on generic oblivious memory primitives, oblivious search and update functionalities were not investigated in this work.

## 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). Our proposed POSUP enables oblivious (single/multi)-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 [65], where we synergize SGX-supported ORAM with Oblivious Data Structure (ODS) [78] to enable oblivious keyword search and update operations on encrypted data. This synergy (*i*) addresses the network bandwidth and communication hurdles of ORAM-SE composition in the client-server setting; (*ii*) eliminates the cost of processing the entire database inside Intel SGX as in [25, 73]; and more importantly, (*iii*) allows for operation on a large outsourced database without being restricted by Intel SGX memory as in [3]. This composition also enables efficient oblivious keyword update capacity. We further outline our contributions as follows:

(1) *New oblivious search and update platform design with SGX:* We construct ODS instantiations for EIDX and EDB by harnessing Intel SGX with Path-ORAM [71] and Circuit-ORAM [76]. POSUP allows for some query types such as single keyword and multi-keyword queries. Moreover, POSUP supports an efficient oblivious update via our optimization tricks that exploit some special characteristics of the underlying oblivious data structures.

(2) *Full-fledged implementation and evaluation:* We implemented POSUP and evaluated its performance on commodity hardware with a large dataset (e.g., a full-size Wikipedia English corpus) containing hundreds millions of keyword-file pairs and millions of files. Our implementation is efficient, taking only 1 ms to obliviously access a 3 KB block with SGX hardware. Our experimental results showed that POSUP incurs much lower end-to-end delay than state-of-the-art solutions as follows:
   - Compared with the ORAM-SE composition in the conventional client-server model (without a secure hardware), POSUP incurs 100× less network bandwidth overhead and and 1000× fewer network communication round-trips. As a result, the end-to-end delay of POSUP is two orders of magnitude lower than that of this approach for both keyword search and update operations (see section 6 for detailed experiments).

– Compared with processing the entire database in Intel SGX (e.g., [25, 73]), POSUP requires less data to be processed by the enclave (i.e. $O(\log N)$ *vs.* $O(N)$, where $N$ is the size of outsourced database). This results in POSUP having 10× lower end-to-end delay than the existing techniques for up to 99.5% of keywords (see section 6). If the number of searched files is small, POSUP can be 100× faster. Moreover, POSUP allows oblivious updating, which does not seem to be fully investigated in state-of-the-art SGX-assisted platforms (e.g., [25]).

(3) *Putting hardware-supported ORAM in real effect:* We take the concept of hardware-supported ORAM primitives to the next level, wherein we develop oblivious data structures and routines with optimizations to provide practical search and update functionalities on large databases, which was not investigated by existing hardware-supported memory primitives (e.g., ZeroTrace). Our implementation will be available at: www.github.com/thanghoang/POSUP.

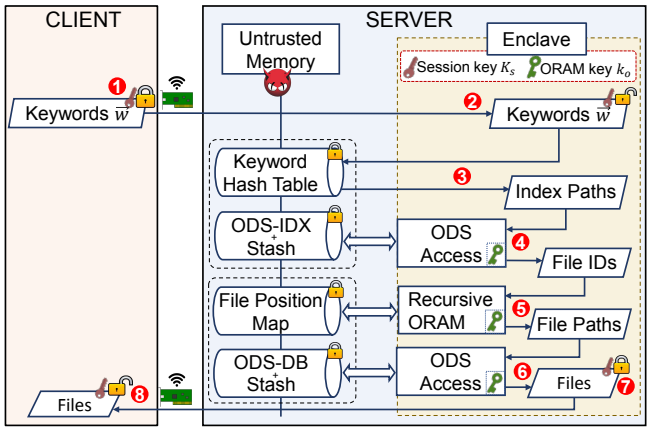



**Fig. 1.** An overview of POSUP workflow. Encrypted data structures (e.g., ODS-IDX, ODS-DB) stored at the server's untrusted memory are created by the client in the *initialization* phase (described in subsection 4.1). ❶–❷: The client encrypts the (search/update) query with session key ($K_s$) and sends it to the enclave. ❸–❻: The enclave performs oblivious accesses (e.g., ODS, recursive ORAM) on encrypted data structure components to retrieve/update files involved with the query. ❼–❽: The enclave encrypts the query results with $K_s$ and sends back to the client.

# 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 commodity secure hardware, as 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 and 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 searches and updates efficiently, even for very large datasets. To defeat attacks against the server's execution logic, POSUP runs its ORAM and ODS controllers in an enclave protected by Intel SGX.

## 2.1 System Model

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

**System Composition.** Figure 1 illustrates the 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 that 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 comprises two parts: an untrusted server and a trusted enclave. (i) The untrusted server provides 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 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 1 outlines the overview of oblivious search and update in POSUP.

• *Initialization*: The client first performs the remote attestation of the 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 the secure channel, the client also sends a key $k_o$ to the enclave, which will be used for ORAM operations.

Upon receiving the key, the enclave on the server will initialize encrypted data structures. Our POSUP is

composed 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 the ODS techniques proposed in [77] to instantiate ODS-IDX and ODS-DB. This data structure initialization step happens only at the first connection. In other words, to perform search and update operations, the client requires only that a session key ($K_s$) be exchanged with the enclave.

• *Oblivious Search/Updates Queries*: ❶ The client encrypts the *search* (resp. *update*) query with the session key $K_s$, and sends it to the enclave (through the untrusted server). ❷ Upon receiving the encrypted query, the enclave decrypts it with $K_s$. ❸ The enclave scans the entire keyword hash table[1] to retrieve block IDs and their location (path) in ODS-IDX that correspond to the query. ❹ If the query is to search, the enclave performs ODS accesses on ODS-IDX using the ORAM key $k_o$ to get matching file IDs. ❺ The enclave determines the location of file IDs in ODS-DB by executing recursive ORAM accesses with ORAM key $k_o$ on the file position map structure. ❻ The enclave performs ODS accesses on ODS-DB with $k_o$ to retrieve file(s) associated with the query. ❼ The enclave encrypts retrieved files with $K_s$ and sends them to the client. ❽ The client recovers encrypted files with $K_s$. The enclave performs the same procedure as search for handling the update query, where it first performs a keyword hash table scan and ODS accesses on ODS-IDX to update blocks in the encrypted index, followed by a recursive ORAM access on the file position map and an ODS access on ODS-DB to update file blocks in the encrypted files.

All these strategies enable us to achieve oblivious keyword search/update operations more efficiently than processing the entire ODS-DB and ODS-IDX in the enclave [25, 73]. Our system is also more efficient than the direct application of existing SGX-ORAM memory primitives [65] because we harness the 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. The client is fully trusted and transfers only the ORAM encryption key $k_o$ to the enclave after establishing a secure channel with the enclave (Therefore, untrusted parts of server cannot obtain this key). To establish this secure channel, we rely on the remote attestation protocol provided by Intel SGX. Thus, we need a trusted authority (right now, it is Intel) for this remote attestation protocol; however, this does not have to be Intel if we utilize a different kind of secure enclave (e.g., Sanctum [18]). We assume the server is untrusted except for the enclave. Specifically, we do not trust any of the server's logic that includes a 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 limitations; it suffers from various side-channel attacks in cache access [10, 30, 32, 49], memory access [11, 81], registers [48], etc [42, 80]. Unfortunately, preventing side-channel attacks against Intel SGX entirely is a very challenging task. Instead of making POSUP side-channel free, we aim only to make POSUP secure against several known side-channel attacks on Intel SGX, which are mentioned above. For simplicity, we do not focus on securing POSUP against size and timing information leakage, which can be easily achieved via padding. We refer the reader to subsection 4.4 for a more detailed discussion.

1 Since keyword universe is arbitrarily large, it is mandatory to maintain a hash table that uniquely matches each keyword to a block ID in the encrypted index for a given dataset (see subsection 4.1 for more details).

# 3 Building Blocks

## 3.1 ORAM

The security of ORAM can be defined as follows.

**Definition 1** (ORAM security [71]). *Let* $\vec{x} = (\mathsf{op}_i, u_i, \mathsf{data}_i)_{i=1}^{q}$ *be a data request sequence, where* $\mathsf{op}_i \in \{\mathit{read}(u_i, \mathsf{data}_i), \mathit{write}(u_i, \mathsf{data}_i)\}$, $u_i$ *is the logical address to be read/written and* $\mathsf{data}_i$ *is the data at* $u_i$ *to be read/written. Let* $\boldsymbol{AP}(\vec{x})$ *be an access pattern observed by the server given a data request sequence* $\vec{x}$.

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

**Path-ORAM.** Stefanov et al. proposed Path-ORAM [71], the most efficient and simple ORAM scheme, which follows the tree paradigm by Shi et al. [66]. In this paradigm, there are three components: (*i*) a complete binary tree (stored at the server) with $N$ leaf nodes that can store up to $N$ data blocks. Each node in the tree is called a *bucket*, which can store up to $Z$ blocks, all of which are IND-CPA encrypted; (*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. Both pos and $S$ are maintained by the client. To access a block with Path-ORAM, the client retrieves its path from pos and then performs a read operation (ReadPath) on its path, in which all real blocks in the path are fetched into $S$. The client updates the retrieved block with a new random path in pos and then performs an eviction operation (Evict) to push the blocks in $S$ back to the read path such that each block resides somewhere in an intersection node between the read path and its assigned path toward the leaf. Notice that the pos component can be stored on the server in the form of smaller ORAMs via recursion [66, 71]. In Path-ORAM, the stash size was proven to be upper-bounded by the security parameter $\lambda$ as $|S| = O(\lambda)$ blocks, which characterizes the overflow probability and statistical security. We describe the detailed algorithms of Path-ORAM in the Appendix.

**Circuit-ORAM.** Circuit-ORAM [76] reduces the circuit complexity of Path-ORAM by minimizing the number of blocks that are involved during read and eviction. Each bucket has meta-data that store the information about real blocks and their path ID.

**Read:** Similar to the original tree ORAM in [66], the client reads all data in the path, but keeps only the block of interest in the stash and removes it from the path. The removal process can be implemented efficiently by flipping only 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. The 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 the 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 the eviction path being selected randomly or deterministically, as in [28] (see the Appendix for details).

Circuit-ORAM incurs approximately $1.25\times$ more I/O accesses than Path-ORAM. However, it has a smaller circuit size, and therefore, it is more efficient to be implemented with Intel SGX [65]. 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 was proven to be upper-bounded by the 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 the 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 to protect communication channels between the enclave and the client. We also implement our logic in the enclave using oblivious primitives in the Intel processor such as `CMOV` and `SETE` instructions to prevent potential access pattern leakage.

As building blocks of POSUP, we utilize the following 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 in 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 stores only encrypted data on it. Therefore, 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' execution).
- *Remote attestation and key exchange*: SGX supports the remote attestation of the enclave to authenticate whether the configuration of the enclave is correct and to share a secret key for secure communication [17, 39] 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; 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 the 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.

• *Privacy concerns on relying on Intel SGX*: The root of trust of Intel SGX relies on an infrastructure provided by Intel (e.g., the Intel Attestation Service and the quoting enclave). Our current implementation uses Intel SGX for its secure enclave; therefore, its privacy is bound to Intel's discretion. However, we believe that this is just an implementation-specific issue and that the use of alternative open-source secure enclaves such as Sanctum [18] on the RISC-V architecture [79] could relax this restriction, e.g., by distributing trust over the Public Key Infrastructure (PKI), similar to how Transport Layer Security (SSL/TLS) works in practice.

• *System calls*: Because an enclave is a part of the user-level process, Intel SGX does not provide any protection on privileged operations such as system calls, e.g., 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 and the OS so the enclave can securely communicate across the security boundary if it is provisioned with a secret key for the encryption; this can be done securely via remote attestation.

**Secure operations inside enclave.** We implement oblivious assignment (oupt) and oblivious equality comparison (ocmp) functions based on `CMOV` and `SETE` instructions proposed in prior works [57, 62], which do not leak access patterns via control-flow side-channel attacks when POSUP executes the 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 prior works [57, 62] for a detailed description of these functions. Note that our ocmp function slightly differs from what was originally proposed in [57], where we employ `SETE` instead of `SETG` instruction for equality checking.

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

**Fig. 2.** OGet function in POSUP.

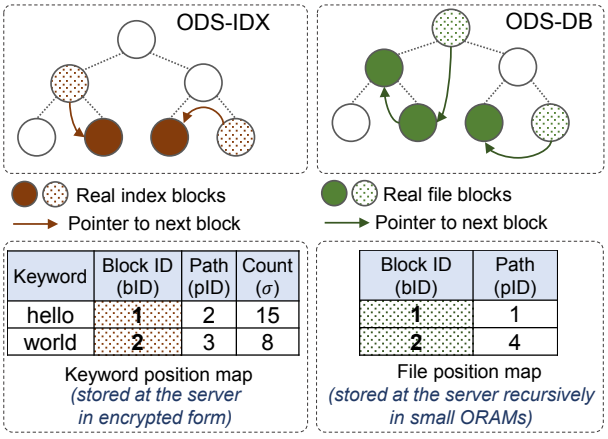



**Fig. 3.** Illustration of ODS-IDX and ODS-DB packaged into ORAM tree in POSUP. Blanked nodes denote dummy blocks, while colored nodes denote real file/index blocks. In colored nodes, pattern-filled nodes denote the head block of linked lists.

ZeroTrace [65] 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 patterns. We implemented our version of OReadPath and OEvict and refer readers to ZeroTrace [65] for a detailed description. In OReadPath and OEvict, we scan the entire stash and path and then use ocmp and oupt to put real blocks from the path to the stash, or vice versa. This results in Path-ORAM being more computation-expensive than Circuit-ORAM as follows. 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. Circuit-ORAM processes only one targeted block at a time and only incurs two separate loops that scan the entire path once to get target blocks and the entire stash. As a result, Circuit-ORAM is more computation-efficient than Path-ORAM when dealing with a large dataset and large block size (see section 6). We then implement the OGet function (Figure 2), which reads a block ID from the stash $S$ into the enclave without leaking access patterns 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 detail.

## 4.1 Oblivious Data Structures

Figure 3 presents the overview of ODS-IDX and ODS-DB in POSUP. ODS-IDX and ODS-DB follow the tree

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 $\vec{\mathsf{id}}_i := (\langle \mathsf{id}_{i_1}, 1\rangle \ldots, \langle \mathsf{id}_{i_n}, 1\rangle)$ are file IDs containing $w_i$
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: $\mathsf{pos}_f[B_{i1}.\mathsf{bID}] \leftarrow B_{i1}.\mathsf{pID}$
11: $\mathcal{B} \leftarrow \mathcal{B} \cup \{B_{i1}, \ldots, B_{im_i}\}$
12: ODS-DB $\leftarrow \mathsf{BuildORAMTree}_{k_o}(\mathcal{B})$
13: $\mathsf{BuildRecursiveORAM}(\mathsf{pos}_f)$
14: **for** each keyword $w_i \in \mathcal{W}$ **do**
15: $\vec{\mathsf{id}}_i \leftarrow \mathsf{IDX}[w_i]$
16: Split $\vec{\mathsf{id}}_i$ to $m'_i$ chunks $(c_{i1}, \ldots, c_{im'_i})$ each of size $|B'|$
17: $B'_{ij}$.DATA $\leftarrow c_{ij}$; $B'_{ij}$.pID $\xleftarrow{\$} [2^{L'}]$ $\forall j \in [m']$
18: **for** $j = 1, \ldots, m'_i - 1$ **do**
19: $B'_{ij}$.NextID $\leftarrow B'_{ij+1}$.bID
20: $B'_j$.NextPath $\leftarrow B'_{ij+1}$.pID
21: $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{B'_{i1}, \ldots, B'_{im'_i}\}$
22: ODS-IDX $\leftarrow \mathsf{BuildORAMTree}_{k_o}(\mathcal{B}')$
23: TW$[w_i] \leftarrow (B_{i1}.\mathsf{bID}, B_{i1}.\mathsf{pID}, |c_{i1}|)$ for each $w_i \in \mathcal{W}$

**Fig. 4.** 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 ($\mathsf{pos}_f$). 14-22: Build ODS-IDX from IDX. 23: Build keyword hash table (TW). All data in ORAM structures (12,22) are encrypted with ORAM key $k_o$. $B, B'$ denote an ORAM block in ODS-DB and ODS-IDX, resp. $L$ and $L'$ denote the height of ODS-DB and ODS-IDX, resp. $|B|$ denotes the size of $B$. $[x]$ denotes $\{1, \ldots, x\}$.

ORAM paradigm in [66] because POSUP harnesses Path-ORAM and Circuit-ORAM as oblivious access cryptographic primitives. We 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.

**Encrypted index.** 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 IDs $\mathsf{id}_{ij}$ 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 [78]) to connect these blocks with each other, where the information of successive blocks is stored in their predecessors. Thus, each block is in the form of $B := \left(\mathsf{bID}, \mathsf{DATA}, \mathsf{NextID}, \mathsf{NextPath}\right)$, where bID is the block ID; DATA $:= (\langle \mathsf{id}_1, \sigma_1\rangle, \ldots, \langle \mathsf{id}_{n'}, \sigma_{n'}\rangle)$ is the block data, which contains a partial list of file IDs (id) as well as their state ($\sigma \in \{0, 1\}$) indicating whether they are added or deleted; NextID and 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.

**Keyword position map.** Since keywords are arbitrary and can be of any length, POSUP maintains a hash table data structure (TW) to map each arbitrary keyword to an ORAM block ID (bID) as well as its path (pID) in ODS-IDX. Additionally, POSUP stores a counter ($\beta_i$) for each keyword in TW to indicate the *actual* number of $\langle \mathsf{id}, \beta\rangle$ pairs that are stored in the head block of the list. So, TW is of the form $\langle \mathsf{key}, \mathsf{value}\rangle$, where key is the hash of the keyword and value contains a triplet (bID, pID, $\beta$). We denote the access operation to the value component in TW as $(\mathsf{bID}, \mathsf{pID}, \beta) \leftarrow \mathsf{TW}[w_i]$.

In POSUP, we maintain TW under the encrypted form in the storage server. This allows the client to be *stateless* and easily extensible to the multi-client setting (see subsection 4.3 for further discussion). Since, to the best of our knowledge, there is no oblivious hash table mechanism, the enclave performs a linear scan on TW for reading/writing component(s) in TW to hide the access pattern. Notice that recursive-ORAM might not be applied on TW because it requires deterministic indexes to operate, while keywords to be accessed are arbitrary.

**Encrypted files.** We apply the same principle as in the encrypted index construction to build 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 ID 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.

**File position map.** POSUP maintains the file position map to keep track of the path of the head block in each ODS linked list. Note that we can index file IDs 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 side. Our design needs to perform only recursive ORAM access to get the path of the starting block, while the path of other blocks in the linked list is obtained from their predecessor due to the ODS technique. This reduces the number of recursive calls on the file position map compared with the direct application of oblivious memory primitives (e.g., [3, 24]) for enabling oblivious query functionality.

We present the detailed algorithm to construct ODS-IDX and ODS-DB in Figure 4. We encrypt ODS-IDX and ODS-DB with ORAM key $k_o$ and store both ODS-IDX and ODS-DB on the server's untrusted

memory. The stash components required by underlying ORAM schemes 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 Update and Search Protocols

Since search operations require us to clear stale data that might arise due to previous update operations, we first introduce the oblivious update protocol in POSUP and then describe the oblivious search.

### 4.2.1 Oblivious Update

For simplicity, we assume that the file to be updated ($f_{\mathsf{id}}$) is already present at the client side. The update operation incurs $f_{\mathsf{id}}$ to be added/deleted/modified from EDB, and some keyword-file pairs to be added/deleted from EIDX. Intuitively, for each keyword ($w_i$) to be updated in $f_{\mathsf{id}}$, the client requests the enclave to add id along with the update state (add/delete) to an empty data slot in the head block of the linked list, which represents the search result of $w_i$ in EDB. If there is no available slot, the enclave picks an empty block[2] in EDB, adds update information into it and then links it with the current head block of the linked list by updating its NextID and NextPath values. This strategy results in blocks close to the head of the list containing the latest updated file IDs. Figure 5 outlines the oblivious update protocol with the following details.

*1) Update ODS-IDX*: The client updates the file content, and forms a list of keywords ($\vec{w}$) to be added or deleted in the updated file ($f_{\mathsf{id}}$) (❶). The client encrypts the update query ($q$) containing id and $\vec{w}$ with $K_s$ and sends it to the enclave. The enclave decrypts $q$ with $K_s$ (❷) and then accesses the entire keyword position map (TW) to retrieve the block ID (bID), path (pID), and current counter ($\beta$) of updated keywords (❸). For each updated keyword ($w_i$), the enclave checks whether there is an empty slot in the head block of $w_i$ in ODS-IDX (❸). If this does not hold, the enclave gets the ID and the path of an empty block in ODS-IDX (❺) and updates this block as the new head of the linked list of $w_i$ in TW (❻). Notice that POSUP performs all comparison and update operations in the oblivious manner using ocmp and oupt functions described in subsection 3.2 to prevent instructional leakage. Once the target block is determined, the enclave performs an ORAM access on ODS-IDX to add id and the update state ($\sigma$) into it (❼–⓫). Specifically, for each updated keyword ($w_i$), the enclave first executes OReadPath with path $\mathsf{pID}_i$ (❼) to

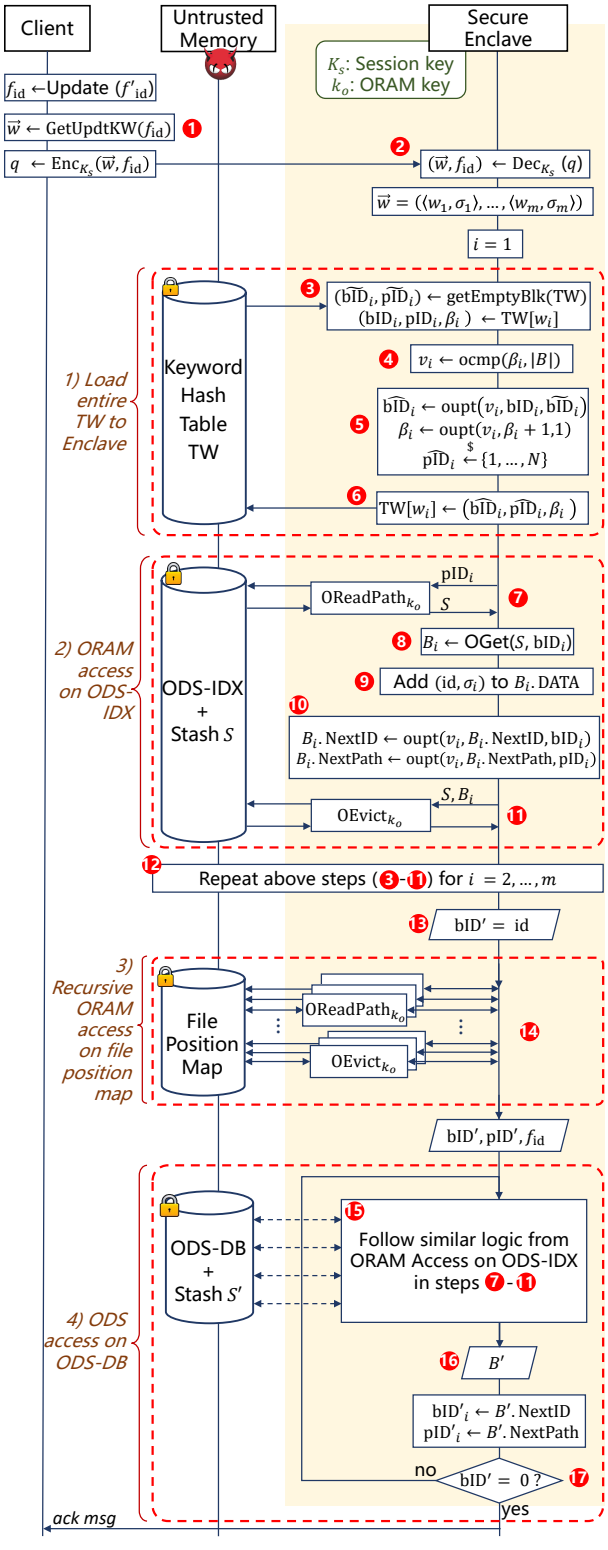



**Fig. 5.** Our oblivious update protocol. ❶-❷: Generate (encrypted) update query. ❸-❻: Access on entire keyword hash table (TW). ❼-⓬: Update encrypted index (ODS-IDX) via ORAM access. ⓭-⓮: Recursive ORAM access on file position map to retrieve location of updated file. ⓯-⓱: Update the file via ODS access on encrypted files (ODS-DB).

**2** ID of empty blocks can be stored in a separate data structure (e.g., list).

fetch the block ($B_i$) of ID $\mathsf{bID}_i$ from ODS-IDX into the stash ($S$). Second, it reads $B_i$ from $S$ via OGet (❽) and then adds $\langle \mathsf{id}, \sigma \rangle$ to the block data ($B_i$.DATA) (❾). If $B_i$ was previously empty, it updates the pointer of $B_i$ to link it with the current block head of $w_i$ (❿). Finally, the enclave performs OEvict to write the updated $B_i$ back to ODS-IDX (⓫).

*2) Update ODS-DB*: Given the ID (id) of the updated file (⓭), the enclave executes recursive ORAM accesses on the file position map to retrieve the location of id in ODS-DB (⓮). The enclave splits the updated file $f_{\mathsf{id}}$ into several chunks and then executes ODS access on ODS-DB to update the ORAM data blocks in the linked list of $f_{\mathsf{id}}$ with these chunks (⓯–⓱).

We can see that the deletion in POSUP does not actually delete some real data in ODS-IDX but instead performs addition with the state bit ($\sigma = 0$). This lazy deletion enables the efficient update, which only requires $O(1)$ access on the encrypted index compared with $O(r)$ in the actual deletion due to the search, where $r$ is the number of files in which the updated keyword appears. The price to pay for this efficiency is the cost of increasing the *actual* size of the search index and the search complexity. To mitigate this impact, after $k$ (system parameter) successive updates, POSUP performs a *dummy* search on the most frequently updated keyword to do garbage collection, since the search operation in POSUP will clear stale data appearing in blocks toward the tail of the linked list, as described in the next section.

### 4.2.2 Oblivious Search

Figure 6 presents the oblivious search protocol in POSUP. First, the client executes the remote attestation protocol with the enclave to establish a secure communication channel with a session key ($K_s$). The client then encrypts the search query of the form $q = w_1 \star_1 \cdots \star_{m-1} w_m$ with $K_s$, where $\star_i \in \{\vee, \wedge\}$ (❶) and sends it to the enclave. The enclave decrypts $q$ with $K_s$ to obtain the list of searched keywords ($w_i$) and performs the following operations.

*1) Access on keyword hash table (TW):* The enclave scans TW entirely to get the block ID ($\mathsf{bID}_i$) and its path ($\mathsf{pID}_i$) in the index (ODS-IDX) of each $w_i$ (❷).

*2) ODS access on ODS-IDX:* For each $\langle \mathsf{bID}_i, \mathsf{pID}_i \rangle$ pair, the enclave performs an ODS access on ODS-IDX (containing multiple ORAM accesses) to retrieve all blocks in the linked list, all of which form the entire list of file IDs ($R_i$) matching $w_i$ (❸–⓫). According to our update strategy (see subsubsection 4.2.1), blocks toward the head of the oblivious linked list will contain file IDs with the *most up-to-date* update state. Hence, during the block update operation (❺), the enclave will

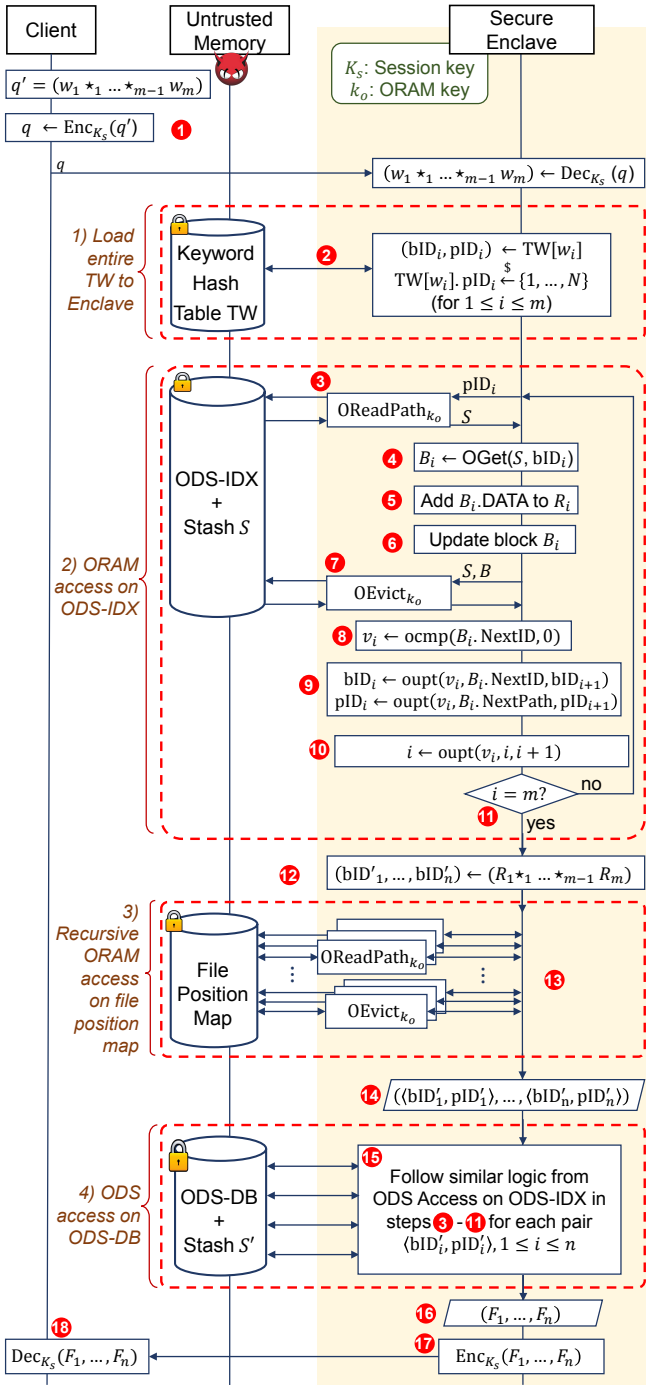



**Fig. 6.** Detailed search protocol. ❶–❷ Generate encrypted search query ($q$). ❸–⓬ ODS access on ODS-IDX to retrieve file IDs matching with $q$. ⓭–⓮ Recursive ORAM access on file position map to get the location of matched files. ⓯–⓲ ODS access on ODS-DB to retrieve all matched files and send to the client.

clear stale file IDs in the accessed block if they are already present in $R_i$, or their most recent update status is "delete." Once the block becomes empty (meaning it does not contain any file IDs left), the enclave will store its ID in the empty list in TWso that it can be re-used

later in the update operation. After the block is accessed with ORAM, the enclave determines if it links with another block via ocmp (8). If this holds, the enclave gets the next block information (9) and continues the oblivious access as above. Otherwise, it processes the next searched keyword (10).

Note that we perform all these conditional checks and processing in an oblivious manner via ocmp and oupt functions. This prevents POSUP from leaking the information regarding size of individual searched keywords (11) but only the total amount of data that are processed due to the search query.

*3) Recursive ORAM access on file position map*: After the 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 matching $q$ (12)[3]. For each block $\mathsf{bID}'_i$ in the joint list, the enclave performs recursive ORAM access (13) on the file position map structure to retrieve the corresponding path $\mathsf{pID}'_i$ of $\mathsf{bID}'$ in ODS-DB (14).

*4) ODS access on ODS-DB*: The enclave performs a sequence of ODS accesses on ODS-DB with the same logic as ODS-IDX accesses to retrieve the file content of each $\mathsf{bID}'_i$ (15–16). Finally, the enclave encrypts all the retrieved files with the session key $K_s$ (17), and sends them to the client for decryption (18).

**Discussion.** One might observe that although POSUP can support boolean queries, the strategy we have applied is simple, in which we conduct an oblivious search on each keyword present in the query first, followed by taking the union/ intersection over all the search results. This trivial strategy is not (sub)optimal and might be inefficient for some complex boolean forms, in which each keyword appears in a large proportion of databases, but the final combination only returns a few results. Nonetheless, our main focus in this paper is to demonstrate the effectiveness of using secure hardware to improve the performance of searchable encryption and ORAM composition in practice. Therefore, we build the POSUP platform by leveraging a simple oblivious data structure in the form of a single linked list, which seems only optimal for single keyword queries and efficient update. We leave the POSUP optimization for other query functionalities (e.g., ranked query, conjunctive query) as open research questions.

Another limitation of POSUP is the linear scan of the keyword hash table in the enclave due to our collision concern when obliviously mapping arbitrary keywords to a finite set (block IDs). To improve this, one might consider using *the power of two choices* hashing technique [64] to minimize the collision probability, thereby enabling (recursive) ORAM operations atop the keyword hash table. We will also leave this as an open research question to be investigated in the future work.

## 4.3 Extension to Multi-user Setting

In POSUP, the client is stateless. Thus, it is easy to extend POSUP into the multi-user setting including a data owner (who owns $n$ outsourced files), a storage server, and $k$ users (who want to search/update on $n$ files) as follows. The data owner creates an access control data structure (ACDS) to grant permission (e.g., search/update) for $k$ users on $n$ files. The data owner encrypts and sends ACDS to the enclave, along with the encrypted index (ODS-DB) and encrypted files (ODS-DB) that are constructed, as described in subsection 4.1, all of which are stored in the server's untrusted memory (e.g., SSD).

Given that a user wants to search for a keyword, he/she will authenticate with the enclave using, for example, the user identifier and password. If authenticated, the enclave performs oblivious access on ODS-IDX (as described in subsubsection 4.2.2) to obtain file IDs matching the query. For each file ID, the enclave accesses ACDS with ORAM to check whether the user has the read permission on the file. If so, the enclave performs oblivious access on ODS-DB to retrieve the file and sends it to the user. The same principle applies to the file update procedure. Roughly speaking, the enclave first authenticates the user and then obliviously accesses ACDS to check whether the user has the update permission on the file. If permitted, the enclave executes the oblivious update protocol as presented in subsubsection 4.2.1.

## 4.4 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 in Definition 1.

In POSUP, we can observe from Figure 5 and Figure 6 that search and update operations incur the same oblivious access procedures on encrypted data structures. In particular, given a search/update query, the enclave first performs (*i*) access on the entire keyword hash table and then, (*ii*) ODS access(es) on ODS-IDX, followed by (*iii*) recursive ORAM access(es) on the file position map, and finally (*iv*) ODS access(es) on ODS-DB. In the update protocol, add and delete operations also invoke the same oblivious access procedure, where they differ from each other only in terms of the state bit

**3** Since the file ID (id) is assigned to the ID (bID) of the head block in the linked list, we use id and bID interchangeably.

value ($\sigma$), which is encrypted in the view of the attacker. Hence, in general, any search/update queries that are of the *same* size and incur the *same* number of (recursive) ORAM and ODS accesses are *indistinguishable*.

**Size information leakage.** Since ORAM and our linked list ODS do not hide the number of oblivious accesses, POSUP might leak the size of the query, which can allow the attacker to distinguish access patterns, thereby learning information about the query. The size information can be inferred in several points when the enclave performs oblivious operations as follows. In the search protocol (Figure 6), the size can be learned by an attacker from (*i*) the search query (❶); (*ii*) the number of accesses on the keyword hash table (❷) and the encrypted index (❶1,❶2); (*iii*) the number of recursive ORAM accesses on the file position map (❶3,❶4) and encrypted files (❶6); (*iv*) and the result returned to the client (❶7). Similarly, in the update protocol (Figure 5), the size can be leaked from the update query (❶,❷), or the number of accesses on encrypted data structures (❶2,❶7).

To mitigate the impact of size leakage, we can apply padding to all aforementioned positions. For instance, for the query that requires less than $n'$ total ORAM accesses, one can add dummy ORAM accesses on both ODS-IDX and ODS-DB, and dummy recursive ORAM accesses on the file position map to quantize the total number to be $n'$, thus making the query size *indistinguishable* by the attacker. We can further apply padding to obfuscate the actual size of the search/update query as well as the size of (search) results being sent from the enclave to the client at the end of the protocol. However, we notice that such padding strategies are generally application-specific, which fully depends on the characteristics of a particular dataset and user preferences, and also might incur heavy bandwidth and processing overhead as the trade-off. This is because padding will increase the cost of oblivious operations less than $n'$ (suppose the number of required operations is $n$) to be equal to that of $n'$ actual operations (where $n' > n$ ). We refer the reader to Ryoan [36] for its parts of data-oblivious communication as well as quantizing processing time to learn more on how the size quantization by padding can thwart such a side channel attack.

**Other side-channel attacks against enclave.** Although the enclave of Intel SGX provides security guarantees such as data confidentiality and integrity against direct memory access attacks, it is not free from side-channel attacks. POSUP does not aim to defeat all sorts of side-channel attacks, which seems to be a very difficult task; instead, we try to build POSUP as a best-effort approach to make it secure against known side-channel attacks. The use of recursive ORAM and ODS in POSUP naturally defeats side-channel attacks on data access patterns such as cache side-channel attacks [10, 30, 32]. Employing oblivious data comparison (ocmp) and oblivious data assignment (oupt) in POSUP (see subsection 3.2) defeats attacks on the control-flow side channel [11, 49, 81] because these primitives eliminate 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. For cryptographic operations 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 an enclave to execute ODS access on ODS-DB and ODS-IDX. As mentioned in section 4, our platform stores ORAM stashes encrypted in the untrusted memory (RAM/SSD), and they are loaded into the enclave when needed.

# 6 Evaluation

We first describe our configuration and evaluation methodology, followed by the main experimental results.

## 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 the 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 the database (DB) is 27 GB (on the disk), and the total size of the search index (IDX) is 6.9 GB. Figure 7 presents the size distribution of text articles in the `enwiki` dataset.

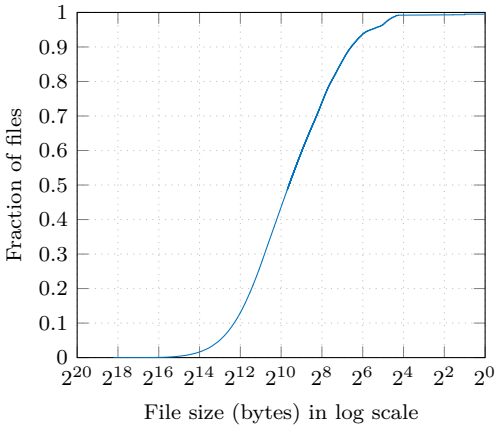



**Fig. 7.** File size distribution in `enwiki` dataset in CDF. An $(x, y)$ point denotes that $y$ fraction of files are sized larger than $x$ bytes. Based on this distribution, we choose the block size of ODS-DB as 3 KB because more than 50% of files are smaller than 3 KB.

**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 then mimic the bandwidth and latency of using Amazon EC2 from our lab. The average network latency and transmission throughput are 18 ms and 150 Mbps, respectively.

**Configurations and Methodology.** We compare POSUP with 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 $\leq 2^{23}$). Because more than 50% of files in our dataset are smaller than 3 KB (shown in Figure 7), 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 appears in less than 512 files (see Figure 8c). We represent a file identifier with a 4-byte integer. This results in ODS-IDX being obliviously accessed up to four times for most search cases. We set the stash size of both ORAM schemes as 80 to achieve negligible overflow probability [71, 76].

• *ORAM-ODS-SE - Direct ORAM-SE composition in a traditional client-server model (without secure hardware).* In this setting, we used the same configuration as in POSUP, where we integrate ODS into the ORAM-SE composition for fair comparison with POSUP. We also set the size of ODS-IDX and ODS-DB to be identical to POSUP. ORAM-ODS-SE differs from POSUP in terms of the ORAM communication channel (over network *vs.* local bus) and the keyword position map location (client *vs.* server). 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 the conventional 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. 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. In other words, EIDX and EDB are loaded and processed (decrypt/encrypt) in 95 MB chunks sequentially inside the enclave. For the update cost, we measured the delay of decryption and re-encryption of the entire EIDX and EDB inside the enclave. Notice that in POSUP, we selected the size of ODS-IDX and ODS-DB that have sufficient empty spaces for later addition of the same amount of dataset size in the setup phase (i.e., 27 GB file with 6.9 GB index). Hence, we double the size of EIDX and EDB in EntireSGX to assume it can also support addition, similar to POSUP, for fair comparison between two techniques.

**Table 1.** Execution time of a single ODS on ODS-IDX and ODS-DB, and recursive ORAM on file position map in POSUP. We ran each operation 500 times and took the average value.

| Operation | Execution Time ($\mu$s) | |
|---|---|---|
| | 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 |

## 6.2 Experiment Results

### 6.2.1 Micro Benchmark

POSUP is efficient, where it takes less than 1 ms 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. Three factors cause 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 an 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 $\mathsf{pos}_f$ is:

$$D_{\mathsf{pos}_f} = \sum_{i=1}^{l} \left( |B| \cdot Z \cdot k \cdot \left( \log_2 \left( \frac{N}{R^i} \right) + 1 \right) \right), \tag{2}$$

where $N$ is the total number of files, $R = |B|/4$ is the compression ratio (assume that a path ID is represented by 4 bytes) and $l = \lfloor \log_R N \rfloor$.

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

**I/O Access.** The I/O access in POSUP is efficient because we implemented the caching technique proposed in [51], 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 520-767$\mu$s to $\leq$285$\mu$s. 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 the file position map is small (i.e., ≈0.2 GB), we store the entire map directly on the RAM. This results in its I/O access delay being negligible (i.e., $\leq$41 $\mu$s).

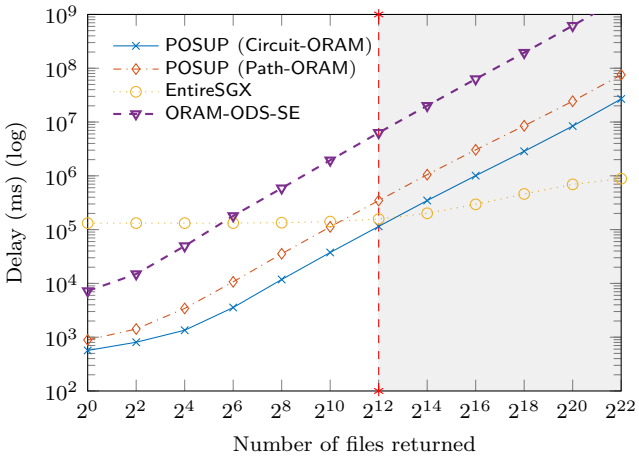
**Enclave Process.** Processing data in the enclave has more effect on the access delay than I/O access because it handles encryption and decryption when reading data from ORAM. Another point that we observed from Table 1 is that the cost of executing the Path-ORAM controller in the enclave is much higher than that of Circuit-ORAM because its read/eviction is more aggressive. Specifically, when using Path-ORAM controller, the enclave must perform $O(\log N) \cdot |S|$ number of encryptions/decryptions, where $|S| = 80$ is the stash size. In contrast, using the Circuit-ORAM controller requires $O(\log N) + |S|$ number of encryptions/decryptions. Therefore, our benchmarked result has shown that integrating Path-ORAM with secure hardware is much less efficient than Circuit-ORAM due to the multiplied factor $|S|$, which is 80. The processing delay of recursive ORAM is high because this requires the enclave to perform additional ORAM encryptions and decryptions on $O(\log N)$ recursion levels.

Table 1 also illustrates that it takes 830 $\mu$s to obliviously access a 512 B block in ODS-IDX with Circuit-ORAM. That is, the latencies of performing single-keyword searches on ODS-IDX in many cases are likely similar to each other, and they are mostly dominated by the number of files to be returned (see subsubsection 6.2.2).

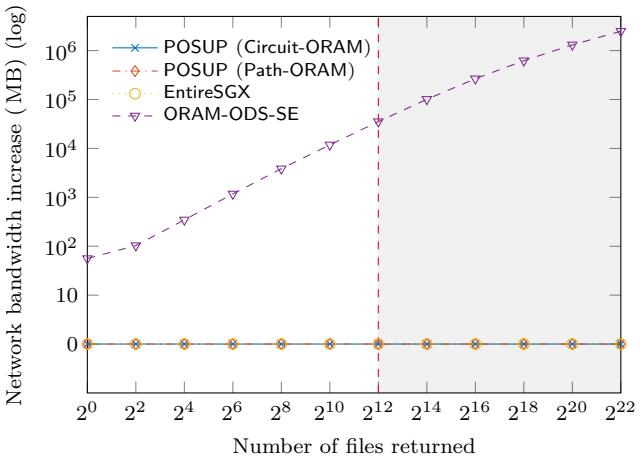
We now illustrate the formula for calculating the number of ODS and recursive ORAM accesses incurred in each search and update query. Given a search query $q$ with $n$ keywords $w_i$, let $m_i$ and $m'$ be the number of files matching $w_i$ and the final $q$, respectively. The search $q$ on POSUP incurs $\sum_{i=1}^{n} \lceil \frac{m_i}{|B|} \rceil$ accesses on ODS-IDX plus $m'$ recursive ORAM accesses on $\mathsf{pos}_f$ 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 $m$ accesses on ODS-IDX plus one recursive ORAM access on $\mathsf{pos}_f$ plus $\lceil \frac{|f|}{|B|} \rceil$ accesses on ODS-DB.

Because the delay in I/O access and encryption in an 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 1. Moreover, as explained in subsection 4.1, each search/update operation in POSUP additionally incurs one-time decryption and re-encryption of the entire keyword hash table (TW), which costs 210 ms for 188 MB-sized TW constructed from the `enwiki` dataset.

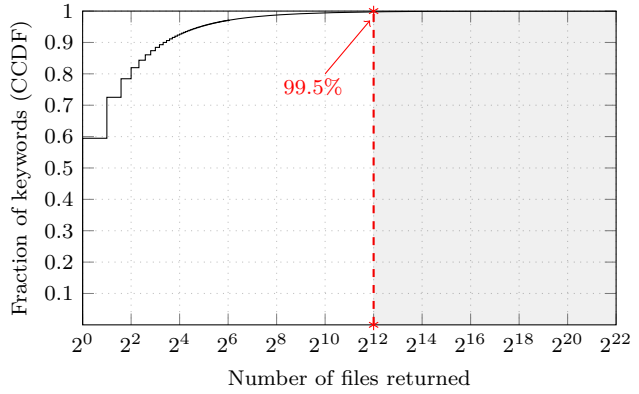
In the following, we present actual benchmarked delay for search and update operations to showcase the efficiency of our system compared with other techniques.



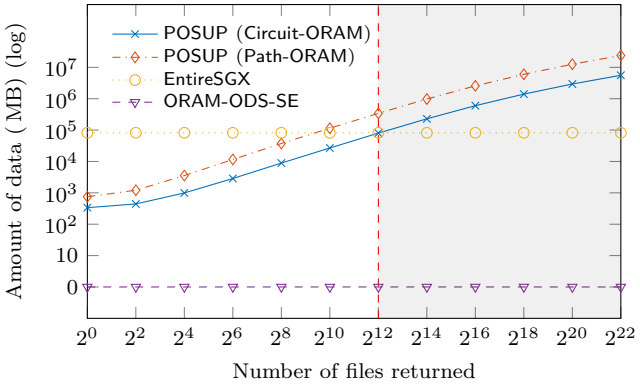

**(a)** End-to-end delay in POSUP and its counterparts regarding how many files are returned in a single-keyword search.




**(b)** Network bandwidth increase of POSUP and its counterparts. Hardware-assisted techniques do not incur network overhead.




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




**(d)** Amount of data being accessed and processed by SGX of POSUP and its counterparts.

**Fig. 8.** 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 18 ms latency. POSUP is more efficient than both EntireSGX and ORAM-ODS-SE for 99.5% of keywords, as indicated by the red dashed lines.

### 6.2.2 Search Delay

POSUP consumes 100× less network bandwidth and requires 1,000× fewer communication round-trips than ORAM-ODS-SE. POSUP incurs 4.5× - 245× less processing time in the enclave than EntireSGX for returning $< 2^{12}$ 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 being up-to 74× and 232× lower than ORAM-ODS-SE and EntireSGX, respectively.

Figure 8a 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 of 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 an enclave, so it does not incur significant network communication overhead like ORAM-ODS-SE, as shown in Figure 8b; 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× more than POSUP) but also in generating a large number of network round-trips (i.e., 1000× more than POSUP) due to multiple rounds incurred in the recursive ORAM and ODS operations. 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% of keywords that can be searched. In Figure 8a, when searching keywords that returns $\leq 2^{12}$ files, POSUP is more efficient than EntireSGX. Figure 8c presents the (accumulative) keyword distribution on `enwiki`. The Zipf's law distribution [56] shown in Figure 8c indicates that the cases returning $\leq 2^{12}$ files are the majority (99.5%), and this indicates that POSUP is

more 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 per ORAM access, while EntireSGX works with the entire index and dataset as its working set, as presented in Figure 8d. For a small fraction of keywords ($< 0.5\%$), 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 require 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 $r \geq \frac{N}{C \cdot k \cdot \log_2 N}$, where $k \in \{8, 10\}$, then processing the entire database in the enclave is better than performing ORAM. Our benchmark result in Figure 8a respects this formula. Theoretically, POSUP should incur more memory accesses than accessing the entire memory when it processes more than $2^{15}$ files. The graph shows that overhead start to become significant when $2^{13}$ files are returned, we can assume the constant $C$ as 4, and then POSUP processes more than $\frac{N}{4 \cdot k \cdot \log_2 N}$ file blocks in the enclave.

**Padding overhead.** Padding so that a search query has the same size as another query will result in a total delay of two queries becoming similar. For example, padding on one-file-involved queries to make their size the same to four-file-involved queries incurs 46% extra delay (239ms) compared with the non-padding case.

### 6.2.3 Update Delay

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

We selected the file with the 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 9 presents the end-to-end update delay of POSUP and its counterparts. POSUP is one order of magnitude faster (40×) than ORAM-ODS-SE because it does not increase bandwidth and round-trip overhead, as analyzed in subsubsection 6.2.2. POSUP with Circuit-ORAM produces the highest throughput so that it achieves the lowest update delay among its counterparts. POSUP is up to 3,300× faster than EntireSGX due to the inevitable overhead in I/O writing required by the design of EntireSGX. 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.

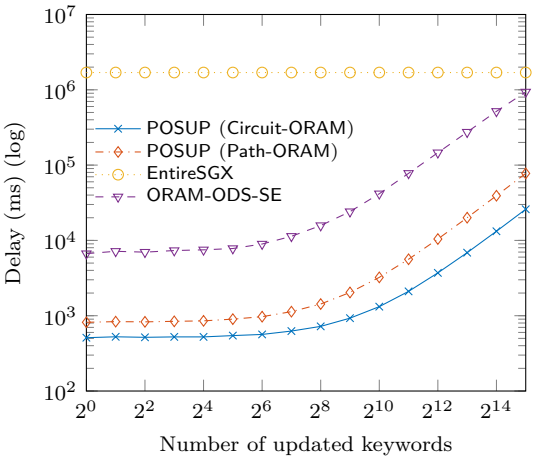



**Fig. 9.** End-to-end delay of updating a 290 KB file with a different number of updated keywords involved.

### 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 $|\mathsf{TW}| + |\mathsf{ODS\text{-}IDX}| + |\mathsf{ODS\text{-}DB}| + |\mathsf{pos}_f| = 0.19 + 34 + 97 + 0.19 \approx 131\,\text{GB}$ if POSUP uses Circuit-ORAM. The corresponding overhead is $0.19 + 68 + 194 + 0.38 \approx 262\,\text{GB}$ if POSUP uses Path-ORAM. Note that some capacity of the ORAM structure is reserved to enable oblivious update (e.g., addition/deletion). Therefore, our server storage overhead presented above can allow the further addition of 3× more IDX and DB presented in subsection 6.1.

## 7 Related Work

**Searching on encrypted data.** Searchable Encryption (SE) [67] enables the client to conduct search operations on encrypted data. Curtmola et al. [19] proposed a secure Symmetric SE (SSE) scheme that supports a single-keyword search, followed by refinements with improved search functionalities (e.g., [12, 72, 75]) and security (e.g., [16, 46]). Kamara et al. [44] proposed a Dynamic SSE (DSSE) scheme that supports update functionality. Afterward, several DSSE schemes proposed offering different features in terms of security (e.g., [9]), efficiency (e.g., [14, 22, 45]), and query functionalities (e.g., [15, 43, 74]). Another line of research focuses on developing encrypted query techniques that are compliant with legacy systems. For instance,

ShadowCrypt [33] and Mimesis Aegis [41, 47] 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 [59] and its variants (e.g., [1, 2, 4, 58, 60]) leverage property-preserving encryptions (e.g., [6, 8]) to perform encrypted structured queries (e.g., SQL) to legacy database management systems (e.g., MongoDB).

**Security vulnerability of encrypted search solutions.** All the aforementioned techniques leak access pattern, which 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 high probability (e.g., [13, 40, 50, 82]). The legacy-compatible encrypted search techniques leak substantial additional information beyond the access pattern because of property-preserving encryption techniques [13, 31, 54, 61].

**Solutions to remedy security vulnerabilities.** ORAM [29] can hide both read and write access patterns, and therefore, it has been considered to seal such leakages in oblivious storage [7, 69, 70] and searchable encryption [26, 53]. Despite its merits, ORAM incurs a poly-logarithmic bandwidth blowup [71, 76], which has been shown to be costly for searchable encryption in the standard client-server network setting [7, 53, 68]. Although ORAMs with constant client-server bandwidth blowups have been proposed recently (e.g., [20, 34]), they either incur higher delay than bandwidth-logarithmic ORAMs because of homomorphic encryption (e.g., [27]), or require multiple computing servers, which increases the deployment cost in practice.

The ORAM communication lower bound has been well-established [29, 76]. 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 [24, 51, 63]. 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 [65] 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 [23]. Eskandarian and Zaharia proposed ObliDB [21], which harnesses Intel SGX and Path-ORAM to enable oblivious SQL queries on database systems. Concurrent with this work, Mishra et al. proposed Oblix [52], an oblivious search and update platform that harnesses Intel SGX, Path-ORAM and oblivious data structures as similar to POSUP.

# 8 Conclusion

In this paper, we developed a new SGX-supported oblivious search and update platform called POSUP. We achieved this by realizing efficient 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 English Wikipedia corpus). The experiments showed that POSUP achieves two to three orders of magnitude less bandwidth blowup and communication round-trips than the conventional client-server models for oblivious search and updates. Similarly, POSUP offers one order of magnitude less processing blowup over alternatives that process the entire outsourced database inside SGX.

# Acknowledgments

We would like to thank all the anonymous reviewers for their insightful comments and suggestions to improve the quality of this paper. This work is partially supported by the NSF CAREER Award CNS-1652389.

# 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. In *Symposium on Network and Distributed System Security (NDSS)*, 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] V. Costan, I. Lebedev, S. Devadas, et al. Secure Processors Part II: Intel SGX security analysis and MIT Sanctum Architecture. *Foundations and Trends® in Electronic Design Automation*, 11(3):249–361, 2017.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[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] Y. Jang. Building Trust in the User I/O in Computer Systems. *Georgia Institute of Technology*, Aug. 2017.

[42] Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)*, Shanghai, China, Oct. 2017.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *Security and Privacy (S&P), 2018 IEEE Symposium on*. IEEE, 2018.

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

[64] A. W. Richa, M. Mitzenmacher, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual, Volume I: Base User-level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[80] 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.

[81] 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.

[82] 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

Figure 10 outlines the general access procedure that tree-based ORAM schemes (e.g., Path-ORAM, Circuit-ORAM) follow. We present the detailed algorithms of Path-ORAM [71] in Figure 11. We give the detailed algorithm of Circuit-ORAM [76] with the deterministic eviction strategy [28] in Figure 12, which execute subroutines in Figure 13.

Tree-basedORAM.Access(op, bID, data*):
1: $x \leftarrow$ pos[bID]
2: pos[bID] $\xleftarrow{\$}$ $[0...2^{L-1}]$
3: $S \leftarrow$ ReadPath($x$)
4: data $\leftarrow$ Read block with ID bID from $S$
5: **if** op = write **then**
6: $S \leftarrow (S - \{(\text{bID}, \text{data})\}) \cup \{(\text{bID}, \text{data}^*)\}$
7: Execute Evict procedure
8: **return** data

**Fig. 10.** General access procedure in tree-based ORAM schemes.

PathORAM.ReadPath($x$):
1: **for** $l = 1, \ldots, L$ **do**
2: $S \leftarrow S \cup$ ReadBucket($P(x, l)$)

PathORAM.Evict:
1: Let $x$ be the read path in PathORAM.ReadPath procedure
2: **for** $l = L, \ldots, 1$ **do**
3: $S' \leftarrow \{(\text{bID}, \text{data}') \in S : P(x, l) = P(\text{pos[bID]}, l)\}$
4: $S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$
5: $S \leftarrow S - S'$
6: WriteBucket($P(x, l), S'$)

**Fig. 11.** Path-ORAM protocol. The ReadBucket($\cdot$) function takes as input a target bucket, reads all blocks in the bucket, and outputs only real blocks.

CircuitORAM.ReadPath($x$):
1: **for** $l = 0, \ldots, L$ **do**
2: **if** (bID, data) $\leftarrow$ ReadAndRm($P(x, l)$, bID) $\neq \perp$ **then**
3: $S \leftarrow S \cup$ (bID, data)

CircuitORAM.Evict:
1: Let $t$ be a global timestamp initialized with 0
2: $x \leftarrow$ order-reversal of base-2 digits of ($t \mod 2^L$)
3: $t \leftarrow t + 1$
4: Execute PrepareDeepest($x$) and PrepareTarget($x$) subroutines to pre-process arrays deepest and target
5: hold $\leftarrow \perp$; dest $\leftarrow \perp$
6: **for** $i = 0, \ldots, L$ **do**
7: towrite := $\perp$
8: **if** (hold $\neq \perp$) and ($i$ = dest) **then**
9: towrite $\leftarrow$ hold
10: hold $\leftarrow \perp$; dest $\leftarrow \perp$
11: **if** target[$i$] $\neq \perp$ **then**
12: hold $\leftarrow$ read and remove deepest block in $P(x, i)$
13: dest $\leftarrow$ target[$i$]
14: Place towrite into bucket $P(x, i)$ if towrite $\neq \perp$
15: Repeat steps 2-14

**Fig. 12.** Circuit-ORAM protocol with deterministic eviction. The ReadAndRm($\cdot, \cdot$) function takes a target bucket and the block ID bID to be accessed (determined in Access function) as input, reads all real blocks from the bucket input, removes the block with ID bID if it appears in the target bucket, and outputs bID along with its data data.

PrepareTarget($x$):

1: dest $\leftarrow \perp$; src $\leftarrow \perp$, target $\leftarrow (\perp, \ldots, \perp)$
2: **for** $i = L, \ldots, 0$ **do**
3: **if** $i =$ src **then**
4: target$[i] \leftarrow$ dest; dest $\leftarrow \perp$; src $\leftarrow \perp$
5: **if** ((dest $= \perp$ and $P(x, i)$ has empty slot) or (target$[i] \neq \perp$)) and (deepest$[i] \neq \perp$) **then**
6: src $\leftarrow$ deepest$[i]$
7: dest $\leftarrow i$

PrepareDeepest($x$):

1: deepest $\leftarrow (\perp, \ldots, \perp)$ src $\leftarrow \perp$; goal $\leftarrow -1$
2: **if** stash $S$ is not empty **then**
3: src $\leftarrow 0$
4: goal $\leftarrow$ Deepest level that a block in the stash $S$ can legally reside on path $P(x)$
5: **for** $i = 1, \ldots, L$ **do**
6: **if** goal $\geq i$ **then** deepest$[i] \leftarrow$ src
7: $\ell \leftarrow$ Deepest level that a block in $P(x, i)$ can legally reside on path $P(x)$
8: **if** $\ell >$ goal **then**
9: goal $\leftarrow \ell$
10: src $\leftarrow i$

**Fig. 13.** Circuit-ORAM eviction subroutines.