Memory-Efficient Encrypted Search Using Trusted Execution Environment

Viet Vo(✉)

Monash University, Melbourne, Australia
Viet.Vo@monash.edu

Abstract. Dynamic searchable encryption (DSE) is important to enable dynamic updates (addition/deletion) on an encrypted database maintained by an untrusted server hosted on the cloud. It is desired that such updates should reveal as less as possible the information revealed to the server. As a result, advanced security notions of forward and backward privacy have been proposed to categorise the leakage by via addition and historical deletion, respectively. However, recent backward-(forward)-private schemes are not efficient enough to support very large databases. In this paper, we resort to the trusted execution environment, i.e., Intel SGX, to ease the above bottleneck. In detail, we proposed Magnus that guarantees Type I^{-} backward privacy. Our key idea is to leverage a compressed Bloom filter within the Intel SGX's enclave to verify the deletion documents with the search keyword. This optimisation minimises the communication overhead between the SGX and untrusted memory. Then, to reduce the enclave's memory, Magnus further relies on a position map-free oblivious data structure maintained by the untrusted server. This improvement is to avoid paging effect in the enclave.

1 Introduction

Dynamic searchable encryption (DSE) [11,18,22] enables users to update/query encrypted database managed by untrusted servers (i.e., cloud) securely while preserving search functionalities.

Recent attacks (e.g., file injection attacks) exploiting the leakage in dynamic operations drive the rapid development of DSE schemes revealing less information while performing updates (i.e., addition/deletion). They are formalised as backward and forward privacy notions in DSE. Newly added data is no longer linkable to queries issued before, and deleted data is no longer searchable in queries issued later. As a result, many backward and forward-private DSE schemes have been proposed [3,5,15,24]. However, we note that they often reduce the efficiency of SE, especially in the communication cost between the client and server. Therefore, recent trusted execution environment-supported schemes have been proposed to accelerate the update/search operations. For example, Amjad et al. [1] proposed Fort, the first forward and Type-I backward private SE schemes using SGX. However, the scheme is still inefficient due to the high I/O complexity between the SGX

X. Yuan et al. (Eds.): QShine 2021, LNICST 402, pp. 340–351, 2021.
https://doi.org/10.1007/978-3-030-91424-0_20

and server. Vo et al. [25] proposed Maiden, the scheme achieves better asymptotic computation. However, we find that the scheme is not memory efficiency although it can achieve Type-I backward privacy. To avoid the memory overhead, we propose Magnus, which can reduce $\mathcal{O}(N)$ memory complexity in the SGX enclave, where N is the database size.

Contributions: Our contributions can be summarised as follows:

- Motivated by the memory bottleneck during Search in Maiden, we design Magnus to achieve forward and Type-I$^-$ backward-private for supporting real document insert/addition. Magnus leverages the SGX enclave to carefully track keyword states and document deletions, in order to minimise the communication overhead between the SGX and untrusted memory. Magnus also employs a Bloom filter to compress the information of deletions, which speeds up the search operations. Magnus leverage oblivious data structures to hide access patterns on the search index and real documents.
- We formalise the security model of our schemes and perform security analysis accordingly.

2 Related Work

Dynamic Searchable Encryption: The seminal work in the field was presented by Kamara et al. [18] proposing a DSE scheme with sublinear search time. Since then, many studies have been proposed to enrich search functionality [28,31] as well as improve the security [4,24].

Forward and Backward Privacy: In dynamic SE, forward privacy blocks the old query tokens on retrieving newly inserted data. It has been used to mitigate file-injection attacks [3,23,30]. Backward privacy [5,24] prevents the adversary from knowing the historial data manipulation of the client (e.g., historical insertion time of deleted data). There are three types of backward privacy, Type-I to Type-III, in the descending order of security.

Encrypted Search with Trusted Execution: Another research direction in the field is to leverage hardware-assisted trusted execution environment (TEE) [1,8,14,19]. In general, TEE such as Intel SGX can improve the communication between the client and server by letting the SGX's enclave play the client's role in token generation (i.e., update/query). For instance, ObliDB [13] and Oblix [19] build up Path-ORAM trees to support insertion and deletion on SQL tables. HardIDX [14], POSUP [17], and BISEN [2] improve the search efficiency in supporting encrypted document search. Note that these work do not support *forward* and *backward privacy*. Hence, they are not relevant to our focus in this paper. Until recently, Amjad et al. [1] proposed three schemes supporting Type-I, II and III backward private SE to enable single-keyword query; they are, Fort, Bunker-B, and Bunker-A, respectively. Then, [26] proposed Type-II SGX-SE1 and SGX-SE2 schemes, which outperform Bunker-B in both search latency and update computation/communication. Very recently, Vo et al. [25] also proposed

Table 1. Comparison with Maiden. N denotes the total number of keyword/document pairs. a_w presents the total number of entries of addition and deletion updates performed on w. n_w is the number of (current, non-deleted) documents containing w, d_w denotes the number of deletions performed on w. D and W denote the total number of documents, and the total number of keywords, respectively.

Type-I scheme	Communication enclave-server			Enclave computation			Enclave storage
	Add	Del	Search	Add	Del	Search	
Maiden	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(WlogD)$ $+\mathcal{O}(a_wW)$ $+\mathcal{O}(N)$
Magnus	$\mathcal{O}(log^2N)$	$\mathcal{O}(log^2N)$	$\mathcal{O}(d_wlogN)$ $+\mathcal{O}(n_wlog^2N)$	$\mathcal{O}(log^2N)$	$\mathcal{O}(log^2N)$	$\mathcal{O}(d_wlogN)$ $+\mathcal{O}(n_wlog^2N)$	$\mathcal{O}(WlogD)$ $+\mathcal{O}(a_wW)$

Maiden, Type I backward privacy, that supports very large deletion and achieves better performance than Fort. However, the scheme is not memory friendly to enclave's due to paging overhead.

3 Background

3.1 Trusted Execution Environment

TEE likes Intel SGX [9] is a set of instructions forming a secure and isolated execution environment. The environment minimises the attack surface to only the CPU processor. Other components are untrusted. The trusted execution part of the application is located in a protected memory area with strong protection enforced by SGX. The untrusted part is executed as a normal process and can invoke the enclave only through a predefined communication interface of ecalls/ocalls. We refer readers to [10] for the security guarantee of Intel SGX, related side-channel attacks [21,29], and recent countermeasures [6,16,20].

3.2 Dynamic Searchable Symmetric Encryption

Here, we briefly overview dynamic SE and the notion of *forward* and *backward privacy* in dynamic SE. We refer readers to [3,5] for formal definition of these security notions. A dynamic SE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ defines the client and a server via following protocols:

$\underline{\mathsf{Setup}}(1^\lambda, \mathsf{DB})$: The protocol inputs a security parameter λ and outputs a secret key K, a state ST for the client, and an encrypted database EDB to the server.

$\underline{\mathsf{Search}}(K, w, ST; \mathsf{EDB})$: The protocol allows to query w by using (K, ST) to generate query token q for w. Upon receiving q, the server follows the protocol to retrieve the search result (i.e., matching document) of q.

$\underline{\mathsf{Update}}(K, (\mathsf{op}, \mathsf{in}), ST; \mathsf{EDB})$: The protocol takes K, ST, an input in associated with an operation op from the client, and EDB, where $\mathsf{op} \in \{add, del\}$ and in

consists of a document identifier id and a set of keywords in that document. Then, the protocol inserts or removes in from EDB upon op.

There are two security notions based on the leakage function of dynamic SE [5]. The *forward privacy* ensures that addition update prevents using the old query token to retrieve that new added data. The *backward privacy* guarantees that when a keyword-document pair (w, id) is added and then deleted, subsequent searches on w do not reveal id. We use the notations in [3,5] to restate Type-I and Type-II *backward privacy* as follows. Let Q be the current query list, and $\mathsf{TimeDB}(w)$ be the access pattern on the non-deleted documents *currently* matching w and the timestamps of inserting them to the database. We denote by $\mathsf{Updates}(w)$ the time stamps of updates on w. Then, formally,

$$\mathsf{TimeDB}(w) = \{(u, id) | (u, add, (w, id)) \in Q \text{ and } \forall (u', del, (w, id)) \notin Q\}$$

$$\mathsf{Updates}(w) = \{(u | (u, add, (w, id)) \text{ or } (u', del, (w, id)) \in Q\}$$

Type-I *backward privacy* is the most secure [5]. It only reveals what time the current (non-deleted) documents matching to w added (i.e.,TimeDB(w)). In contrast, Type-II reveals both $\{\mathsf{DB}(w), \mathsf{Updates}(w)\}$ to the untrusted server.

4 Our Proposed Scheme

In this section, we present the system design for Magnus, our assumption, and threat model. Then, we investigate the limitation of previous TEE-supported Type-I *backward-private* scheme (i.e., Maiden), and highlight our design intuition. Afterwards, we detail the protocols of Magnus.

4.1 System Overview

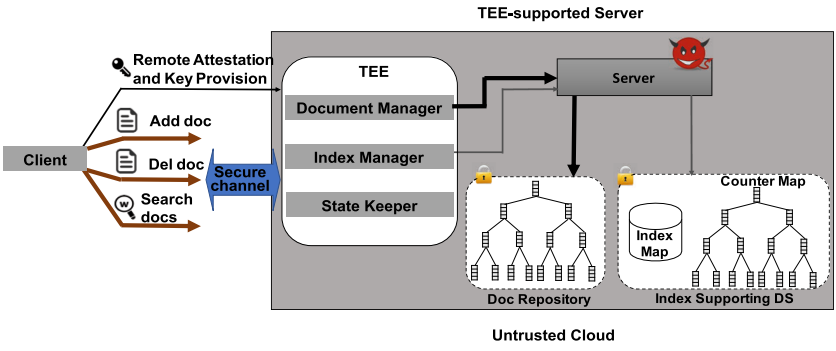

Fig. 1. System design for Type-I$^-$ *backward-private* SE

In our system (see Fig. 1), the *Client* can remotely manipulate the database via Setup, Update (add/del documents), and Search operations. The *Server* manages

the encrypted document repository R, and encrypted supporting data structures of index map M_I and counter map M_c. We design R and M_c using oblivious data structure maps [27] to support oblivious accesses. The *Enclave* is inside the *Server*, and it contains necessary components of *Document Manager* (resp. *Index Manager*) to perform data (resp. index) queries to the untrusted parts.

In Setup, the *Client* remotely authenticates the *Enclave* via attestation protocol [9]. Then, she establishes a secure channel with the *Enclave*.

In Update, we design the *Client* to be storage-free. Giving a document doc with an unique identifier id, the *Client* sends a tuple $(\mathsf{op} = add, \mathsf{in} = \{\mathsf{doc}, id\})$ to the *Enclave* via the established secure channel. Then, *Document Manager* parses the doc to generate encrypted data blocks and obliviously insert them to R. During that step, *Document Manager* internally updates the local *State Keeper*. To support index search, *Index Manager* generates addition tokens for $\forall(w, id)$ in doc and obliviously insert to M_I and M_c. If Update is document deletion, the *Client* send a tuple of $(\mathsf{op} = del, \mathsf{in} = \mathsf{doc}', id)$ to the *Enclave*, where doc' is a dummy document to hide the document deletion operation, and id is the real document identifier of the doc to be deleted. Similarly, *Document Manager* and *Index Manager* perform the same process as in document addition, except for additionally local updates within the *Enclave*.

In Search, the *Client* sends a query keyword w to the *Enclave*. Accordingly, the *Index Manager* executes the Search protocol of Magnus to only retrieve the currently matching documents of w. Then, *Document Manager* reconstructs these docs from querying encrypted data blocks from R. At the end of the Search operation, the *Client* receives the docs in a batch manner.

4.2 TEE Assumptions and Threat Model

Our Assumptions with TEE: We assume that TEE like SGX *Enclave* behaves correctly, without hardware bugs or backdoors), and the code and data inside the enclave are protected. Also, we assume the communication between the *Client* and the *Enclave* relies on the secure channel created during SGX attestation. Like many other hardware-supported works [12,19], we consider side-channel attacks [7,21,29] against SGX are out of our scope. In this paper, we only focus on how the efficiency of *forward* and *backward* privacy. Denial-of-service (DoS) attacks on the Intel *Enclave* and the server are also out of our focus.

Threat Models: We consider a semi-honest but powerful attacker at the server-side. She can gain full access over software stack outside of the enclave, OS and hypervisor, as well as hardware components in the server except for the processor package and memory bus. In particular, the attacker can observe memory addresses and timestamps when accessing (encrypted) data on the memory bus, in- memory, or in EDB to generate data access patterns.

4.3 Design Intuition

In this paper, we focus on the trade-off between the enclave's memory overhead and Type-I backward privacy. In this leakage type, we have Fort [1] and

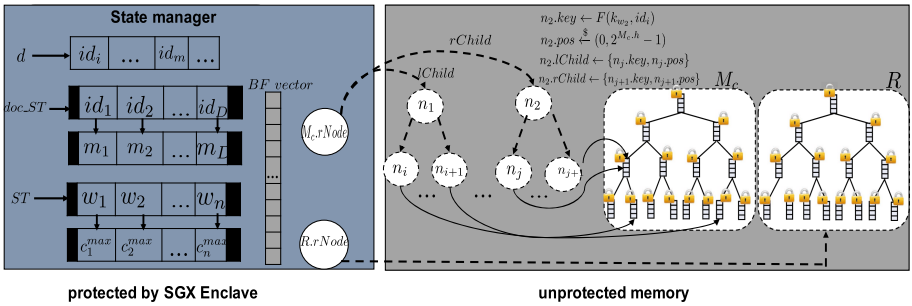

Fig. 2. AVL tree stored in Path-ORAM

Maiden [25], which are TEE-supported Type-I *backward-private* SE scheme. However, we only analyse the practical limitation of Maiden since it is more efficient than Fort. We note that Maiden requires a large enclave's storage (i.e., $\mathcal{O}W(logD) + \mathcal{O}(a_w W) + \mathcal{O}(N)$ to achieve Type-I. In details, the scheme needs $\mathcal{O}W(logD)$ to store keyword state ST, another $\mathcal{O}(a_w W)$ for Bloom filter checking, and an important $\mathcal{O}(N)$ overhead to store the states of document identifiers, i.e., $(id_i, c+i)$, mapping to w. In this way, it only needs the *Server* to store the index map M_I. However, the scheme is not memory efficient since the *Enclave*'s memory is limited at 98 MB. Exceeding usage causes paging effect in Intel SGX. Therefore, we propose Magnus, the scheme only requires $\mathcal{O}W(logD) + \mathcal{O}(a_w W)$ memory overhead in the enclave. As a trade-off, the scheme only achieves Type-I$^-$ backward privacy. The reason is that, Magnus additionally introduces a new leakage that only happens during Search operations. In particular, that is ORAM accesses to the *Server* during the operation, leaking the number of deleted documents of the query keyword w (i.e., d_w) when the *Server* records the number of visited ORAM positions.

4.4 Magnus Construction

Underlying Data Structures: Magnus utilises a key/value oblivious data structure (ODS) to build a state map, namely M_c, by using an AVL tree [27]. The AVL nodes are stored in a non-recursive Path-ORAM, where each node n_i contains the information of key/value itself and the meta data of its children nodes, i.e., $n_i = (key, value, pos, height, lChild, rChild)$, where key is an evaluation of $PRF(k_w, id)$, $value = \mathsf{Enc}(k_c, c)$ (dedicated keys k_w, k_c derived by w), pos indicates the node's leaf position in the Path-ORAM, $height$ presents the node's level, $lChild$ is a map of $(lChild.key, lChild.pos)$, and as similar with $rChild$ (see Fig. 2). With the help of this *pointer-based technique*, Magnus does not store he position map pm of $OMAP_c$ in the *Enclave*.

We note that the map lookup for an AVL node in $OMAP_c$ is reduced to $\mathcal{O}(logN)$ ORAM accesses with one for each Path-ORAM level. Hence, Magnus only spends $\mathcal{O}(d_w logN)$ round trips between the *Enclave* and the *Server* to

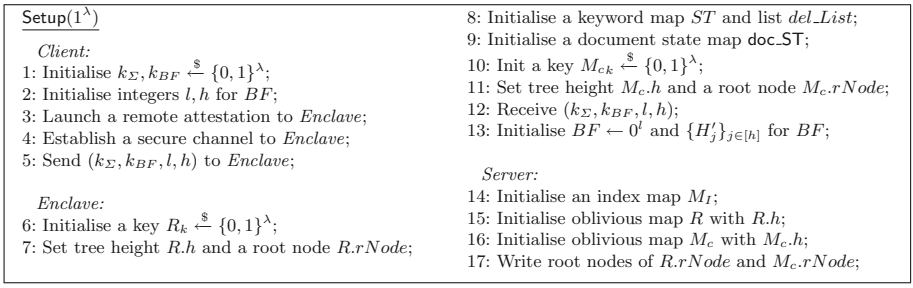

Fig. 3. Setup protocol in Magnus where the *Client* is *storage-free*

retrieve d_w lookup. (see Search communication in Table 1). Upon retrieving AVL nodes in buckets from the *Server*, Magnus employs a negligible *stash* to cache AVL (w, id) nodes in the *Enclave*. We note that the max *stash*'s size is about 147 blocks (∼57 KB) for $Z = 4$ and the failure probability $f_p < 2^{-128}$. Then, the *Enclave* can make updates to these nodes locally (i.e., insertion and rebalancing). Before writing them back to the $OMAP_c$, fetched nodes are assigned new random *pos*, and their parents are also updated correspondingly. We also note that the $OMAP_c$ structure stored at the *Server* is bounded to $Z \cdot 2^{\Sigma(w,id)}$ blocks, to support $\Sigma(w, id)$ entries of *addition* Update. We also make use of the ODS to store the physical blocks of encrypted documents in R. We present the protocols in Magnus as follows.

In Setup (see Fig. 3), the *Client* performs an attestation protocol with the *Enclave* and provisions $K = (k_\Sigma, k_{BF})$ upon an established secure channel, where k_Σ is used to generate update/query tokens and k_{BF} is the key for computing the digest of $(w||id)$, and l presents the vector size of the BF and h is the number of hash functions. The *Enclave* initiates R_k to later encrypt document data buckets, and set the tree height $R.h$. Note that, the document root node $R.rNode$ should always be stored and updated within the *Enclave* to allow remotely traversing the R stored in the *Server*. The *Enclave* initiates a Bloom Filter BF based on the provided (l, h). Then, it also maintains the maps ST and doc_ST, and the list *del_List*, where ST maintains the state of keywords, doc_ST keeps track the number of data blocks for each doc with identifier id, and *del_List* records the deleted *id*s during *deletion* Update. The *Server* holds an encrypted index M_I, the oblivious maps M_c and R.

In Update, the *Client* sends a tuple $(\mathsf{op}, \mathsf{in})$ to the *Enclave* via the secure channel, where $(\mathsf{op} = add, \mathsf{in} = (\mathsf{doc}, id))$ or $(\mathsf{op} = del, \mathsf{in} = (\mathsf{doc}', id)))$. We note that doc' is a dummy document, and it enables the *Enclave* to perform insertion in R even when $\mathsf{op} = del$. After that, the *Enclave* splits the doc to data chunks $\mathsf{B} = \{b_i\}, \forall i \in \{1, m\}$, where b_i is the *value* in an AVL node n_i (see Fig. 4 lines 9–13). There is a multiple round trips $\mathcal{O}(log^2 N)$ between the *Enclave* and the *Server* to insert n_i, via $R.\mathsf{Access}(\mathsf{op}' = update, \mathsf{in}' = (R.rNode, R_k, n_i))$, as presented in Fig. 2. In short, the *Enclave* obliviously traverses from the AVL root $R.rnode$ to find and insert a matching node n_i via the Path-ORAM structure

Update(op, in)

Client:
1: **if** op = *add* **then**// in=(doc,*id*)
2: send (op, in) to *Enclave*;
3: **else** // op = *del*
4: Create dummy doc in doc′;
5: send (op, in = (doc′, *id*)) to *Enclave*;
6: **end if**

Enclave:
7: **if** op = *add* **then** //in = (doc, *id*)
8: Split doc into chunks B = $\{b_1, \ldots, b_m\}$;
//create nodes from chunks
9: **for** $b_i \in$ B **do**
10: $pos \xleftarrow{\$} (0, 2^{R.h} - 1)$;
11: $n_i \leftarrow (key = F(k_\Sigma, id||i), value = b_i, pos, height = 1, lChild = \varnothing, rChild = \varnothing)$;
12: R.Access(op′ = *update*, in′ = $(R.rNode, R_k, n_i)$) to *Server*;
13: **end for**

14: doc_ST[*id*] ← *m*;
15: Parse doc to $\{(w, id)\}$;
16: $T \leftarrow \{\emptyset\}$;
17: **for** (w, id) **do**
18: $(k_w \parallel k_c) \leftarrow F(k_\Sigma, w)$;
19: $c \leftarrow ST[w]$;
20: $c \leftarrow c + 1$;
21: $k_{id} \leftarrow H_1(k_w, c)$;
22: $(u, v) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id))$
23: add (u, v) to T;
24: $pos \xleftarrow{\$} (0, 2^{M_c.h} - 1)$;
25: $n_c \leftarrow (key = F(k_w, id), value = \mathsf{Enc}(k_c, c), pos, height = 1,$ //leaf node $lChild = \varnothing, rChild = \varnothing)$;
26: M_c.Access(op′ = *update*, in′ = $(M_c.rNode, M_{ck}, n_c)$) to *Server*;
27: $BF[H'_j(k_{BF}, w \parallel id)] \leftarrow 1$ for $j \in [1, h]$;
28: $ST[w] \leftarrow c$;
29: **end for**
30: send T to *Server*; // in batch

31: **else:** // op = *del*, in = (doc′, *id*)
32: $r, r' \leftarrow$ rand() from doc′;
33: Create dummy index node $n_i, \forall i \in [0, r]$;
34: M_c.Access(op′ = *update*, in′ = $(M_c.rNode, M_{ck}, n_i)$) to *Server*;
35: Create dummy data node $n'_i, \forall i \in [0, r']$;
36: R.Access(op′ = *update*, in′ = $(R.rNode, R_k, n'_i)$) to *Server*;
37: add *id* to *del_List*;
38: del doc_ST[*id*];
39: set dummy entries in T to *Server*;
40: **end if**

Server:
41: receive T from *Enclave*;
42: **for** (u, v) **in** T **do**
43: $M_I[u] \leftarrow v$;

Fig. 4. Update protocol in Magnus

of R. The ORAM accesses from the *Enclave* are executed via *ocall*s, and downloaded buckets are decrypted by R_k. Visited nodes in the buckets are un-packed and cached in the *stash*. During the traverses, the *Enclave* scans the *stash* first, and perform oblivious accesses if the node is not found. To hide the data-dependent path of the ORAM currently accessing *pos*, the *Enclave* also accesses a dummy ORAM path generated from a random *pos*′. That is, the *Enclave* always retrieves buckets of two ORAM paths, one for real and another one for dummy path, from the *Server*, for every AVL node access. Magnus tries to put the node as deep as possible, i.e., $height = 1$, as a leaf node. Once the node is inserted, its parent node is re-balanced, recursively to the root. Finally, all accessed nodes are mapped to new ORAM positions, and ORAM buckets containing them are re-encrypted with R_k before being stored again at R. Note that Magnus only need 1 *ocall* to send all the new buckets to the *Server* in a batch. After data blocks in doc inserted, the *Enclave* will parses the doc to retrieve a list of $\{(w, id)\}$ and later update M_I and M_c (see line 14). To do so, the *Enclave* utilises the latest state $c \leftarrow ST[w]$ of w, and with $(k_w||k_c)$ generated by $F(k_\Sigma, w)$ to generate an encrypted entry $(u, v) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id))$ in a temporary list T. Note that, H_1 and H_2 denote hash functions and Enc is a symmetric encryption cipher. The entry holds the mapping between c and *id* to allow the *Enclave* to retrieve *id* upon u and the known state. Then, the *Enclave* generates an AVL state node n_c of (w, id) with $n_c.key = F(k_w, id)$ and $n_c.value = (\mathsf{Enc}(k_c, c))$

Search(w)

Client:
1: send w to *Enclave*;

Enclave: //retrieve currently matched *ids*
2: $(k_w \parallel k_c) \leftarrow F(k_\Sigma, w)$;
3: $st_{(w,c)} \leftarrow \{\emptyset\}, Q \leftarrow [\emptyset]$;
4: **for** id **in** *del_List* **do**
5: **if** $BF[H'_j(k_{BF}, w \parallel id)]_{j\in[h]} = 1$ **then**
6: $u' \leftarrow F(k_w, id)$;
7: $v' \leftarrow M_c$.Access(op$'$ $= search$, in$'$ $= (M_c.rNode, M_{ck}, u'))$ in *Server*;
8: $c \leftarrow \mathsf{Dec}(k_c, v')$;
9: $st_{(w,c)} \leftarrow \{c\} \cup st_{(w,c)}$;
10: **end if**
11: **end for**
12: $st_{(w,c)} \leftarrow \{0, \ldots, ST[w]\} \setminus st_{(w,c)}$
13: **for** c **in** $st_{(w,c)}$ **do**
14: $(u, k_{id}) \leftarrow (H_2(k_w, c), H_1(k_w, c))$;
15: $Q \leftarrow \{(u, k_{id})\} \cup Q$;
16: **end for**
18: send Q to *Server*;

Server:
17: receive Q from *Enclave*;
18: *id_List* $\leftarrow \{\emptyset\}$; // currently matched ids
19: **for** (u, k_{id}) **in** Q **do**
20: $id \leftarrow \mathsf{Dec}(k_{id}, M_I[u])$;
21: *id_List* $\leftarrow \{id\} \cup$ *id_List*;
22: send *id_List* to *Enclave*;

Enclave://retrieve currently matched docs
23: **for** id **in** *id_List* **do**
24: $m \leftarrow$ doc_ST$[id]$;
25: **for** i **in** $[0, m]$ **do**
26: $key \leftarrow F(k_\Sigma, id||i))$;
27: $n_i \leftarrow R$.Access(op$'$ $= search$, in$'$ $= (R.rNode, R_k, key))$ in *Server*;
28: $res \leftarrow value(\mathsf{Dec}(R_k, n_i))$;
29: **end for**
30: Return *res* to *Client*; //in batch of docs
31: **end for**

Fig. 5. Search protocol in Magnus

(see line 25 in Fig. 4). In this way, the *Enclave* can retrieve the state c of w if the keyword is in the deleted doc with id in Search. Then, the state node n_c is obliviously inserted to M_c via the Access protocol, in a similar way with R. The enclave also computes a new member of $H'_j(k_{BF}, w \parallel id)$ to update BF. After all, the *Enclave* sends a batch of T to the *Server* within 1 *ocall* per a document *addition* Then, the *Server* will update M_I by using T. If Update is *deletion*, the *Enclave* generates dummy AVL nodes and inserts them to both M_c and R to hide the operation. Then, it updates *del_List* by the deleted id.

In Search for a keyword w, Magnus verifies the mapping between w and deleted id by testing the membership (w, id) with BF. If it is a case, the *Enclave* performs a look-up in the map M_c with the key (w, id) via ORAM accesses (see Fig. 5). Note that we can relax M_c to *write-only* ODS, where ORAM accesses in Search do not require writing new ORAM buckets, similar with Fort. Since the Path-ORAM of M_c is non-recursive and it has $\mathcal{O}(logN)$ levels, the communication round trips to retrieve all state nodes of deleted (w, id) pairs is $\mathcal{O}(d_w logN)$ for d_w deleted ids. The *Enclave* also performs padding with dummy accesses to ensure the *Server* sees a fixed number of ORAM buckets visited per an AVL node ($\approx$1.45$logN$). Then, Magnus retrieves the deleted state list $st_{w_c} = \{c_{id}^{del}\}$, where c_{id}^{del} is the state used for deleted ids. After that, the *Enclave* discards them from the list $\{0, \ldots, ST[w]\}$ to infer the states of non-deleted documents (see Fig. 5 line 12). Then, it computes query tokens in Q and send them to the *Server* in a batch. Upon receiving currently matched *id_List* from the *Server*, the *Enclave* can perform the same deterministic ORAM accesses to retrieve data blocks of **doc**s in *id_List* based on doc_ST$[id]$ (see lines 23–31).

5 Security Analysis

The security of Magnus relies on the black-box use of oblivious data structures (ODS) M_c and R, as initiated in [27]. Magnus contains the leakage of Update and Search operations. We formulate the leakage of Magnus and define $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and a $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ game for an adaptive adversary $\mathcal{A}$ and a polynomial time simulator $\mathcal{S}$ with the security parameter λ as follows.

Let $\mathcal{L}$ be a stateful leakage function $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch})$, where the first three functions define the information exposed to the *Server* in Update, and Search, respectively. In Setup, Magnus leaks the data structure of M_I (i.e., the encrypted index), M_c (i.e., the OMAP of keyword states), R (i.e., the OMAP containing encrypted document blocks). In $\mathsf{Update}(\mathsf{op} = \{add, del\}, \mathsf{in})$, Magnus leaks the data access pattern T_{M_I} of encrypted entries to be inserted in M_I, ORAM positions accessed in M_c and R, denoted as T_{M_c} and T_R, respectively. Then, $\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}) = \{(T_{M_I}, T_{M_c}, T_R)\}$. In $\mathsf{Search}(w)$, Magnus leaks 1) the accessed ORAM positions on M_c, named $\mathsf{ap}_{M_c}(w)$, 2) the leakage $\mathsf{TimeDB}(w)$ when the *Enclave* queries n_w, named $\mathsf{ap}_{M_I}(w)$. Then, formally $\mathcal{L}^{Srch}(w) = \mathsf{ap}_{M_c}(w) + \mathsf{TimeDB}(w)$.

Definition 1. *Consider* Magnus *scheme that consists of three protocols* Setup, Update, *and* Search. *Consider the probabilistic experiments* $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ *and* $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, *whereas* $\mathcal{A}$ *is a stateful adversary, and* $\mathcal{S}$ *is a stateful simulator that gets the leakage function* $\mathcal{L}$.

$\mathbf{Real}_{\mathcal{A}}(\lambda)$: $\mathcal{A}$ takes (1^{λ}) and returns two different computable databases DB^0 and DB^1. Then, the challenger runs the $\mathsf{Setup}(\mathsf{DB}^b)$ upon a chosen bit $b \in \{0, 1\}$. Then, $\mathcal{A}$ makes a polynomial number of Updates (addition/deletion) with $(\mathsf{op}, \mathsf{in})$, where Z is a natural number of documents, and $(\mathsf{op} = add, in = \mathsf{doc}_i)$ or $(\mathsf{op} = del, \mathsf{in} = id_i)$. Accordingly, the challenger runs those updates with $\mathsf{Update}(\mathsf{op}, \mathsf{in})$ and eventually returns the encrypted DB to $\mathcal{A}$. After that, $\mathcal{A}$ adaptively chooses the keyword w (*resp.*, $(\mathsf{op}, \mathsf{in})$) to search (*resp.*, update). In response, the challenger runs $\mathsf{Search}(w)$ (*resp.*, $\mathsf{Update}(\mathsf{op},\mathsf{in})$) and returns the transcript of each operation. Upon receiving the transcript, $\mathcal{A}$ outputs a bit b.

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ chooses a $\mathsf{DB} = \{\mathsf{doc}_i\}_{i \in Z}$. By using $\mathcal{L}^{Updt}$ and $(M_I, M_c, R)^{Updt}$, $\mathcal{S}$ creates a tuple of (M_I, M_c, R) and passes it to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively chooses the keyword w (*resp.*, $(\mathsf{op}, \mathsf{in})$) to search (*resp.*, update). The challenger returns the transcript simulated by $\mathcal{S}(\mathcal{L}^{Srch}(w))$ (*resp.*, $\mathcal{S}(\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}))$) with $(M_I, M_c, R)^{Srch}$. Finally, $\mathcal{A}$ returns a bit b.

Theorem 1. *Assuming OMAPs* M_c *and* R *are created with the secure oblivious map of [27], SGX Enclave are secure, and the communication between the Client and the Enclave is secure,* Magnus *is an adaptively-secure SSE scheme with* $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, \mathsf{in}) = \mathsf{op}$ *and* $\mathcal{L}^{Srch}(w) = \{\mathsf{TimeDB}(w), ap_{M_c}(w)\}$.

Throughout the operations, the *Server* observes a sequence of Path-ORAM positions of M_c and R, for each position is chosen uniformly at random. In addition, the map M_I always get entry inserts during the doc *addition/deletion.*

During Search, Magnus reveals n_w during the query on M_I. In addition, d_w is revealed during the communication round trips between the *Enclave* and the *Server*. This shows that Magnus has Type-I$^-$ leakage, i.e., Type-I *backward privacy*.

6 Conclusion

In this paper, we leverage the advance of Intel SGX to design a Type-I$^-$ backward private dynamic searchable encryption scheme. We carefully analyse the limitation of Maiden and propose new design to avoid memory bottleneck of the SGX enclave.

References

1. Amjad, G., Kamara, S., Moataz, T.: Forward and backward private searchable encryption with SGX. In: EuroSec 2019 (2019)
2. Borges, G., Domingos, H., Ferreira, B., Leitão, J., Oliveira, T., Portela, B.: BISEN: efficient boolean searchable symmetric encryption with verifiability and minimal leakage. In: IEEE SRDS 2019 (2019)
3. Bost, R.: $\Sigma o\varphi o\varsigma$ - forward secure searchable encryption. In: ACM CCS 2016 (2016)
4. Bost, R., Fouque, P.A.: Thwarting leakage abuse attacks against searchable encryption - a formal approach and applications to database padding. Cryptology ePrint Archive, Report 2017/1060 (2017). https://eprint.iacr.org/2017/1060
5. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: ACM CCS 2017 (2017)
6. Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., Sadeghi, A.R.: DR.SGX: automated and adjustable side-channel protection for SGX using data location randomization. In: ACSAC 2019 (2019)
7. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software grand exposure: SGX cache attacks are practical. In: WOOT 2017 (2017)
8. Christian, P., Kapil, V., Manuel, C.: EnclaveDB: a secure database using SGX. In: IEEE S&P 2018 (2018)
9. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Archive (2016)
10. Costan, V., Lebedev, I., Devadas, S.: Sanctum: minimal hardware extensions for strong software isolation. In: USENIX Security 2016 (2016)
11. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: ACM CCS 2016 (2016)
12. Duan, H., Wang, C., Yuan, X., Zhou, Y., Wang, Q., Ren, K.: LightBox: full-stack protected stateful middlebox at lightning speed. In: ACM CCS 2019 (2019)
13. Eskandarian, S., Zaharia, M.: ObliDB: oblivious query processing for secure databases. In: Proceedings of the VLDB Endowment (2019)
14. Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.-R.: HardIDX: practical and secure index with SGX. In: Livraga, G., Zhu, S. (eds.) DBSec 2017. LNCS, vol. 10359, pp. 386–408. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61176-1_22
15. Ghareh Chamani, J., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: ACM CCS 2018 (2018)

16. Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., Costa, M.: Strong and efficient cache side-channel protection using hardware transactional memory. In: USENIX Security 2017 (2017)
17. Hoang, T., Ozmen, M.O., Jang, Y., Yavuz, A.A.: Hardware-supported ORAM in effect: practical oblivious search and update on very large dataset. In: PET 2019 (2019)
18. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM CCS 2012 (2012)
19. Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: an efficient oblivious search index. In: IEEE S&P 2018 (2018)
20. Oleksenko, O., Trach, B., Krahn, R., Martin, A., et al.: Varys: protecting SGX enclaves from practical side-channel attacks. In: USENIX ATC 2018 (2018)
21. Shinde, S., Chua, Z.L., Narayanan, V., Saxena, P.: Preventing page faults from telling your secrets. In: ACM AsiaCCS 2016 (2016)
22. Song, D., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE S&P 2000 (2000)
23. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable symmetric encryption with small leakage. In: NDSS 2014 (2014)
24. Sun, S.F., Yuan, X., Liu, J., Steinfeld, R., Sakzad, A., Vo, V., et al.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: ACM CCS 2018 (2018)
25. Vo, V., Lai, S., Yuan, X., Nepal, S., Liu, J.K.: Towards efficient and strong backward private searchable encryption with secure enclaves. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021. LNCS, vol. 12726, pp. 50–75. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78372-3_3
26. Vo, V., Lai, S., Yuan, X., Sun, S.-F., Nepal, S., Liu, J.K.: Accelerating forward and backward private searchable encryption using trusted execution. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 2020. LNCS, vol. 12147, pp. 83–103. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57878-7_5
27. Wang, X.S., et al.: Oblivious data structures. In: CCS 2014 (2014)
28. Wu, S., Li, Q., Li, G., Yuan, D., Yuan, X., Wang, C.: ServeDB: secure, verifiable, and efficient range queries on outsourced database. In: IEEE ICDE 2019 (2019)
29. Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: USENIX Security 2014 (2014)
30. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: USENIX Security 2016 (2016)
31. Zuo, C., Sun, S.-F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11099, pp. 228–246. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98989-1_12