Eurus: Towards an Efficient Searchable Symmetric Encryption with Size Pattern Protection

Zheli Liu, Yanyu Huang, Xiangfu Song, Bo Li, Jin Li, Yali Yuan, and Changyu Dong

Abstract—To achieve efficiently search and update on outsourced encrypted data, dynamic searchable symmetric encryption (DSSE) was proposed by just leaking some well-defined leakages. Though small, many recent works show that an attacker can exploit these leakages to undermine the security of existing DSSE schemes. In particular, an attacker can exploit even seemingly harmless size pattern to perform severe attacks. Many exiting schemes resort to oblivious RAM (ORAM) to hide search/access pattern; however, even such powerful cryptographic primitive cannot protect size pattern leakage. In this paper, we first show that size pattern can lead to more information leakages, which is not well studied or protected by existing schemes. We then extend the existing privacy notion for DSSE to capture the size pattern leakage, achieving a strong forward and backward privacy definition. Following the definition, we propose a new DSSE scheme Eurus. Eurus can eliminate search/access pattern by relying on a multi-server ORAM scheme, meanwhile reducing size pattern with reasonable efficiency. We show that Eurus can reduce leakage significantly with better efficiency, compared with state-of-the-art leakage reduction schemes.

Index Terms—searchable encryption, size pattern, Oblivious RAM, leakage reduction

1 Introduction

There is an increasing trend to host applications, such as databases [1], e-mail, and file systems etc., in a third-party cloud system. In those applications, keyword search is a widely used functionality. Data encryption is used for protecting privacy; however, once data is encrypted, how to securely perform keyword search queries on encrypted data becomes a practical challenge. Searchable symmetric encryption (SSE) was introduced [2] as a cryptographic primitive that allows a client to perform search queries over encrypted data on untrusted servers. The dynamic version of SSE (DSSE) [3] [4] further enables the client to perform updates after the data outsourced to the servers.

Most DSSE schemes build an encrypted invert index extracted from the outsourced data to speed up search efficiency. When adding/deleting a file, the associated index is updated as well to maintain correctness; meanwhile, some privacy notation is achieved. In practice, an inverted index in the form of a pair (*key*, *value*) is commonly used, where *key* is a keyword, *value* is a list that contains the identifiers of files that contain the keyword. When performing a keyword search, the server first performs a search operation on the index and then returns the identifiers of the matching files to the client. The client can fetch the files based on the search result.

- *Zheli Liu, Yanyu Huang and Bo Li are with the College of Cyber Science, College of Computer Science, Tianjin Key Laboratory of Network and Data Security Technology, Nankai University, China.*
  E-mail: liuzheli@nankai.edu.cn, onlyerir@163.com, boli@mail.nankai.edu.cn.
- *Xiangfu Song is with the School of Computer Science and Technology, Shandong University, Jinan, China. E-mail: bintasong@gmail.com*
- *Jin Li is with the School of Computer Science, Guangzhou University, China. E-mail: jinli71@gmail.com.*
- *Yali Yuan is with Institute of Computer Science, University of Goettingen, Goettingen, Germany. E-mail: yali.yuan@informatik.uni-goettingen.de*
- *Changyu Dong is with the School of Computing, Newcastle University, Newcastle Upon Tyne, U.K. E-mail: changyu.dong@newcastle.ac.uk*

Corresponding author: Xiangfu Song

1.1 Leakages and Attacks of DSSE

To achieve acceptable efficiency, practical DSSEs trade the security by leaking certain information. The leakages [5] are normally modeled as access pattern, search pattern, and size pattern. In particular, access pattern exposes that *where, when, and how often* the encrypted indexes are accessed during keyword search queries and entity update queries. The search pattern refers to whether the client performs search queries to the same keyword or not. The size pattern can be leaked during keyword search queries and entity update queries, including the number of the files in the search result, the number of indexes in the added to or deleted files, and so on.

Some works show that these standard SSE leakages can cause severe attacks. Specifically, the leakage-abuse attacks [6], [7], file-injection attack [8], and count attack [7] exploit access pattern leakage, search pattern leakage and size pattern leakage, respectively. The attackers, knowing some backward information of the underlying dataset, can recover the query and even the encrypted data efficiently. More recent attacks were proposed recently [9], [10], [11], [12], those works either aim to improve keyword recovering rate [10], [11], [12] or perform attack under restricted leakage profiles [9], [11]. Although some SSE schemes [13] [14] [15] [16] [17] [18] [19] have been proposed to prevent the information leakage by achieving *forward* and *backward privacy*, they can only break the linkability between search operations and update operations (i.e., *search-update* linkability).

The *forward privacy* refers to the case that when a new file is added, no keyword information of the newly added file is leaked; *backward privacy* refers to the case that the deleted file cannot be leaked in the keyword search queries after the file has already been deleted.

While forward and backward privacy can efficiently mitigate attacks that exploit update leakage, they are still unable to resist those that use search/access/size pattern. E.g., forward private SSE schemes can only resist the adaptive version of the file-injection attack. However, the non-adaptive file-injection attack that exploits access pattern still works on forward private SE (see [8] for more details). Towards attacks using size pattern, Cash et al. [7] proposed Counter Attack. In particular, in Counter Attack, the adversary is assumed to know some information of the underlying dataset. The idea is that if the number of the matched document is unique, then the adversary can immediately identify which keyword is being queried by the unique size; otherwise, any keyword with that size pattern are candidates. Then the IKK Attack [6] is further used to recover the query by access pattern, more concretely, the search result overlaps between different queries. The Counter Attack is more efficient and accurate than the IKK Attack; it can recover almost all queries efficiently for Enron dataset. The attack exploiting search pattern was firstly studied by Liu et al. [20]. Recently, some attacks using search pattern were proposed for range and k-NN query [11], [12]. Those attacks aim to improve plaintext recovery rate without pre-fixed query distribution. There are also recent attacks that purely exploit size pattern leakage [21], [22], [23], [24], [25]. In summary, forward and backward private SSE schemes are still vulnerable to many leakage-abuse attacks.

Some existing countermeasures. The first approach to reduce searchable encryption leakage is through advanced cryptographic techniques, e.g., Oblivious RAM and Private Information Retrieval (PIR). While using these techniques in a black-box way can obtain higher security, the main concern is efficiency. Hoang et al. [26], [27] proposed an oblivious searchable encryption scheme by leveraging write-only ORAM and PIR to hide search/access pattern efficiently, which requires a linear search complexity due to the use of PIR.

Another approach for leakage-reduction is resorting to trusted hardware, e.g., Intel SGX [28]. Schemes based on trusted hardware can be very efficient, and many searchable encryption schemes [29], [30], [31], [32], [33], [34] based on trusted hardware were designed recently. However, there are many challenges for such schemes as they may not resist many side-channel attacks [35], [36], [37].

1.2 Ignored Size Pattern Leakage

Some SSE schemes [4] [17] [39] [40] are inspired from the oblivious random access machine (ORAM) [4] [40] [41] [42] [43] [44] to achieve high security goal by preventing search/access pattern leakage, where the data block is defined as the (*key*, *value*). However, existing ORAM-inspired SSE schemes [39] [4] [31] [17] [45] did not protect size pattern leakage, which means during search query, the result size of search query is always leaked.

To better illustrate the dangers of size pattern leakage, apart from the search-update linkability, we define two types of linkability as shown in Figure 1: (1) *search-search linkability* to denote that keyword search queries for the same keyword can be identified; and (2) *update-update linkability* to denote that updates to the same file can be identified.

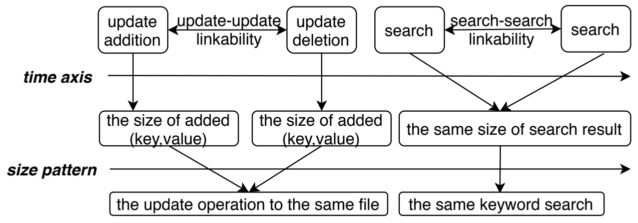

update addition
update-update linkability
update deletion
search
search-search linkability
search
time axis
the size of added (key,value)
the size of added (key,value)
the same size of search result
size pattern
the update operation to the same file
the same keyword search

Fig. 1: The ignored size pattern in general SSE scheme.

Search-search linkability. A SSE scheme with search-search linkability means that the attacker can identify search queries to the same keywords among massive queries. There are several ways of leaking such linkability. For example, if search tokens are linkable in some way, the server can match queries easily, which is nowadays the main source of leakage of search pattern in most SSE schemes. Another possible leaking approach is from memory accessing pattern, i.e., the server can link queries together by observing whether they touch the same memory during search queries. However, things are more tricky since the linkability may also be leaked from the size pattern of matching document result. In particular, if the size pattern of a specific keyword is unique, then the server can easily link queries by only observing their size pattern; this is not an exaggeration – the attack in [7] showed that the unique size pattern is exploited to significantly improve attack efficiency, which turns out to recover almost all queries for Enron dataset.

Update-update linkability. If storage locations of keywords of a file are not protected, the linkability with the previous addition operation is easily leaked when it is deleted. However, simply concealing the access pattern during the update operation is not enough since the size pattern can be leaked from the bandwidth cost of the update query. The linkability refers to the leakage of which file is related to which queries, for example, by observing the resulting lists of multiple queries after a file is added or deleted.

Motivation. For a long time, the size pattern leakage is deemed as harmless leakage. It is until the Counter Attack [7] that shows the size pattern can be further exploited to improve the recovery rate of IKK attack [6]. However, at that time, it was unknown if any attack that purely uses size pattern exists. The situation has changed recently as several new leakage-abuse attacks were proposed. In particular, Blackstone et al. [21] proposed attacks that rely on much weaker assumptions by only exploiting document size information, which can be applied not only to SSE with standard leakage but also to ORAM. Similarly, several attacks [22], [23], [24], [25] to encrypted database that only exploit size pattern leakage were also proposed. All these attacks emphasize that even seemingly harmless size pattern can be used to perform severe attacks.

TABLE 1: Comparison with prior works. N: the number of keyword-file pairs in the database; K: the number of distinct keywords; F: the number of files; n_w: the size of the search result set for keyword w; a_w: the number of entries historically inserted to the database matching keyword w; d_w: the number of deleted files matching w; n_w=$a_w - d_w$; (In practice, $N > a_w > (K, F) > (n_w, d_w)$) ×: break the linkability or seal the pattern. ✓: allow the linkability or the pattern. FP: forward privacy; BP: backward privacy.

Scheme	Computation	Communication	Access pattern		Linkability		
			search	update	search-search	search-update	update-update
Traditional DSSE schemes							
$\Sigma o \phi o \varsigma$ [13]	$O(a_w)$	$O(n_w)$	✓	✓	✓	FP	✓
Fides [16]	$\widetilde{O}(a_w)$	$\widetilde{O}(a_w)$	✓	✓	✓	FP&BP	✓
dual [38]	$O(a_w)$	$O(n_w)$	✓	✓	✓	FP&BP	✓
ORAM-inspired DSSE schemes							
SPS14 [4]	$O(\min\{\begin{matrix} a_w + \log^2 N \\ n_w \log^3 N \end{matrix}\})$	$O(n_w + \log N)$	×	✓	✓	FP	✓
TWORAM [39]	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(n_w \log N + \log^3 N)$	×	×	✓	FP	✓
S^3ORAM [40]	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(n_w \log N)$	×	×	✓	FP	✓
ORION [17]	$O(n_w \log^2 N)$	$O(n_w \log^2 N)$	×	×	✓	FP&BP	✓
HORUS [17]	$O(n_w \log d_w \log N)$	$O(n_w \log d_w \log N)$	×	×	✓	FP&BP	✓
Hoang et al. [26], [27]	$\widetilde{O}(N)$	$\widetilde{O}(\lambda)$	×	×	×	ODSE	×
This work							
Eurus	$O(F^2 \log^2 K)$	$O(max\{n_w, F\} + K))$	×	×	×	strong FP&BP	×

In conclusion, to prevent attacks that exploit SSE leakages, we need to leverage techniques that can eliminate not only search/access pattern but also the size pattern, which then can break the linkability between queries. In this work, we try to design an SSE scheme that achieves the goal with reasonable computational and communication overhead.

1.3 Contributions

We focus on how to protect size pattern in DSSE scheme and define a new security goal by extending the existing forward and backward privacy notion, called *strong FP&BP*, to capture the size pattern hiding. The search-update linkability, search-search linkability, and update-update linkability can be broken under the design of DSSE scheme achieving *strong FP&BP*.

We design the first DSSE scheme Eurus that can achieve *strong FP&BP* with practical performance. We design Eurus upon a multi-server ORAM scheme [40], but we adapt it for our purpose. In Eurus, search/access pattern is eliminated by ORAM simulation while the size pattern is reduced by storage padding. Unlike prior schemes that use black-box ORAM both for search and update, Eurus divides the storage into two parts: one part for black-box ORAM and another called *update slots* for data updating. Those update slots significantly simplify update operation compared with ORAM operation. Besides, oblivious data access is also optimized in Eurus. A detailed comparison can be found in Table 1, SPS14 [4] based on the layer ORAM achieves forward and backward privacy. TWORAM [39] based on the tree ORAM makes use of garbled circuits to reduce the communication frequency but increase the bandwidth cost; however, it cannot achieve backward privacy. The $\Sigma o \phi o \varsigma$ [13] and other schemes can achieve high performance but cannot achieve *strong FP&BP*, compared with our scheme. The experiment results show that, when compared with ORAM-inspired SSE schemes, our scheme is around 46% better for search query over *Enron e-mail* dataset. Moreover, Eurus has a fast update operation due to the use of update slots, which is 4.73× faster than general ORAM-inspired schemes (see Section 7 for more details of the experiments).

2 Preliminaries

Let $\lambda \in \mathbb{N}$ denote the security parameter and negl(λ) denote a negligible function in the security parameter. We assume that all the algorithms take λ implicitly as input. Let $\{0,1\}^l$ be the set of all the l-length binary strings, and $\{0,1\}^*$ be the set of all the finite-length binary strings. Let $|Y|$ denote the cardinality of a finite set Y. Let $y \leftarrow x$ denote the output y of an algorithm x.

2.1 DSSE Framework

In typical DSSE schemes, keywords and files are related by file/keyword pairs. A database can be denoted as DB=$\{(ind_i, W_i)\}_{i=1}^n$, where each $ind_i \in \{0,1\}^l$ is a file identifier, each $W_i \subseteq \{0,1\}^*$ is a set of keywords matching file ind_i and n is the number of files in DB. The set of all keywords in DB is $W = \bigcup_{i=1}^n W_i$. The set of files containing a keyword w is DB(w)= $\{ind_i | w \in W_i\}$. A dynamic searchable symmetric encryption scheme is a tuple of three polynomial-time protocols $\prod = \{Setup, Search, Update\}$:

- (σ, EDB) ← *Setup*(λ, DB): It is a probabilistic algorithm that takes as input the initial database DB, where EDB is an encrypted database and σ is the client state.
- ((σ', DB(w)); EDB′) ← *Search*((σ, w); EDB): It is a client-server protocol, where the client takes as input the state σ and a keyword w, the server takes as input the encrypted database EDB. After the protocol execution, the client outputs an updated state σ' and the set DB(w) of the files (identifiers) containing the keyword w. The server outputs an updated encrypted database EDB′.
- (σ', EDB′) ← *Update*((σ, *op*, ind); EDB): It is a client-server protocol, where the client takes as input the state σ and a file identifier ind, the server takes as input the encrypted database EDB. As a result, the server outputs an updated encrypted database EDB′. If *op* is to add, the updated EDB′ includes the file with identifier ind; If *op* is to delete, the EDB′ is updated by excluding the file with the identifier ind.

Adaptive security. In general, we focus on semi-honest, adaptive and probabilistic polynomial-time (PPT) adversaries for DSSE schemes. *Adaptive* means the adversary can perform queries adaptively, i.e., issuing queries after observing the result of previous queries.

The adversary should learn nothing except for information explicitly allowed to leak. This is captured by the leakage function $\mathcal{L} = (\mathcal{L}_{Setup}, \mathcal{L}_{Search}, \mathcal{L}_{Update})$. The components corresponds to *Setup*, *Search*, and *Update* operations, respectively. An adversary $\mathcal{A}$ tries to distinguish a real-world experiment SSE_{real} and an ideal-world experiment SSE_{ideal} as follows:

- SSE_{real}: the SSE scheme is executed honestly. The adversary observes the real transcript of all the operations, including *Setup*, *Search*, and *Update*.
- SSE_{ideal}: The adversary sees a simulated transcript in place of the real transcript. The simulated transcript is generated by the PPT algorithm $\mathcal{S}$, known as a simulator, that has access to the leakage function $\mathcal{L}$.

Eventually, $\mathcal{A}$ outputs a bit 0 (i.e., SSE_{real}) or a bit 1 (i.e., SSE_{ideal}). The general security model captures the fact that whenever the client triggers one of these operations, the adversary learns no more than the output of the corresponding leakage function.

Definition 1. [*Adaptive security of DSSE scheme*]: The DSSE scheme Π with a collection of leakage functions $\mathcal{L}$ is $\mathcal{L}$ *-adaptively-secure*, if for any PPT adversary $\mathcal{A}$, there exists a simulator S such that the following equation holds:

$$|\Pr[SSE_{real}{}^{\Pi}_{\mathcal{A}}(\lambda) = 1] - \Pr[SSE_{ideal}{}_{S,\mathcal{A},\mathcal{L}}(\lambda) = 1]| \leqslant negl(\lambda).$$

Here, the leakage function $\mathcal{L}$ will keep the query list $\mathcal{Q}$ of all the keyword search and update queries as state :

- *Search query*. It is in the form of (i, w), where i is the sequence number (an index starting at 0 and increasing with every query), and w is the searched keyword.
- *Update query*. It is in the form of (i, *op*, ind) for an update query, where *op* is either add or delete and ind is the identifier of the newly updated file. The data block uploaded to the database is a pair (w, ind), a keyword/identifier pair.

3 Privacy Definition Capture Size Pattern

The allowed leakages in general DSSE schemes can be defined according to the primary query types in the SSE model [5] [13] [18], including the search query and update query. Before presenting the definitions, we first define some necessary notations.

Leakage functions. For the keyword search query, the search pattern $sp(w)$ is denoted as a function of keyword w as follows:

$$sp(w) = \{i : (i, w) \in \mathcal{Q}\}.$$

Thus, sp leaks which search queries match the same keyword.

For the update query including the addition and deletion operations, we use $TimeDB(w)$ as described in [16] to denote full list of all files that matches w, excluding the deleted ones, together with the timestamp of when they were inserted in the database. Formally, $TimeDB(w)$ is constructed from the query list $\mathcal{Q}$ as $TimeDB(w) = \{ (i, ind) | (i, add, (w, ind)) \in \mathcal{Q}$ and $\forall\, i', (i', del, (w, ind)) \notin \mathcal{Q}\}$, where *add* denotes an addition operation, *del* means the deletion. The *TimeDB* captures a strong notion of backward privacy revealing only the time of file insertion currently containing the search query for keyword w.

3.1 Prior definition of forward and backward privacy

Forward privacy. The original definition of forward privacy [13] means that when a new file is added, no keyword information of the newly added file is leaked. The definition is as follows:

Definition 2. [*Forward privacy*]: A $\mathcal{L}-$*adaptively-secure* SSE scheme is forward secure iff there exists a leakage function $\mathcal{L}'$ such that its $\mathcal{L}_{Update}$ can be written as:

$$\mathcal{L}_{Update}(op, W, ind) = \mathcal{L}'(op, |W|, ind),$$

where op denotes the operation type, $|W|$ denotes the number of keywords for newly updated file, ind denotes the identifier of newly updated file.

In summary, the update query cannot leak more than the operation, the identifier and the number of keywords to the newly added file.

Backward privacy. It means that deleted files cannot be leaked in the keyword search queries, i.e., only the information about the existing (not yet deleted) files can be leaked. The highest-level backward privacy in [16] is defined as follows:

Definition 3. [*Backward privacy*]: A $\mathcal{L}$-adaptively-secure SSE scheme is insertion revealing backward-private iff the search and update leakage function $\mathcal{L}_{Search}$, $\mathcal{L}_{Update}$ can be written as:

$$\mathcal{L}_{Update}(op, W, ind) = (op, ind),$$
$$\mathcal{L}_{Search}(w) = (\mathbf{TimeDB}(w), a_w),$$

where a_w is the number of entries historically inserted to the database matching keyword w, and $\mathbf{TimeDB}(w)$ reveals all search result, and the time point each of them was inserted into the database.

What this highest-level backward privacy can achieve is that if a file containing w has been deleted before issuing a search query for w, the server should not learn that the deleted file contains w from the subsequent search queries.

3.2 Strong forward and backward privacy

The current existing FP and BP schemes enhance the standard SSE security by reducing the leakage from the update. However, they still allow the search/access/size pattern to be leaked during the search. Specifically, even the highest-level backward privacy still allows $\mathbf{TimeDB}(w)$ to be leaked, from which the adversary knows all related search result and when each identifier was inserted into the database, which means the adversary can still learn access pattern (from the search result), size pattern (from $|\mathbf{TimeDB}(w)|$) and search pattern (from the overlap of the search result). To counter leakage-abuse attacks, we need to eliminate search, access, and result pattern. To this end, we define strong forward and backward security (referred to as *strong FP & BP*) as follows.

Definition 4. [*Strong FP&BP*]: A *$\mathcal{L}$-adaptively-secure* SSE scheme is strong forward and backward secure iff $\mathcal{L}_{Update}$ and $\mathcal{L}_{Search}$ only leak the following leakage:

$$\mathcal{L}_{Update}(op, w, ind) = (\perp),$$

$$\mathcal{L}_{Search}(w) = (F).$$

In summary, $\mathcal{L}$ satisfies strong FB & BP if the following conditions hold: (1) the update leaks nothing (2) the leakage function for search only includes F – the maximal number of matched files, i.e., $F = \max_{w \in W} |DB(w)|$.

Comparison with prior definitions. Since we remove search/access/size pattern from the leakage function, compared with normal FB & BP, the privacy improvement of *strong FP&BP* can be summarized as follows: (1) for a search query, the search pattern, access pattern, and size pattern all should be sealed; (2) for an update query, the identifier should be hidden, the deletion and addition operations should be indistinguishable; (3) the linkability among all (search/update) queries should be broken. In our strong FP&BP definition, only the maximal number of identifiers matched with a keyword is leaked during the search. Concealing these patterns and linkability from the adversary is the key to resist existing leakage-abuse attacks.

Attentions on the leakage from file operations. In this paper, we mainly focus on the security and leakage of the index itself because this is the main source of leakage. Protecting leakage on index-level suffices for applications that only return the identifiers but not the actual files. For applications that return actual files, we still need to handle the leakage during file retrieval phrase to achieve stronger FP and BP as a whole. In this case, all files should be encrypted and accessed by ORAM. Note that ORAM can only conceal search and access pattern, not the size pattern. In order to hide size pattern during file retrieval phrase, the client has to pad all files to the same length and perform dummy read to make the total number of ORAM access to be the same for different queries.

3.3 Oblivious RAM

The ORAM schemes [44] [46] [40] [41] [47] [48] aimed to hide data access pattern. There are mainly two traditional paradigms of ORAM structure: (1) layer-based ORAM [48] and (2) tree-based ORAM [44]. The tree-based ORAM has smaller communication and computation cost in shuffle operation for write obliviousness than the layer-based ORAM based on the research in [49]. We describe the basic operations in the tree-based ORAM [44] as follows, which is presented in Figure 2.

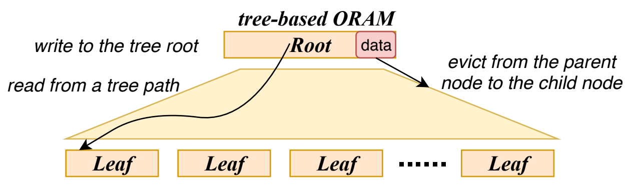


Fig. 2: The basic operations in the tree-based ORAM.

Data access. ORAM should hide whether an operation is read or write when accessing data as well as the address being accessed. One data access contains the following two sub-operations:

- **Oblivious read**. Each data block is assigned to a path in the tree. When the user wants to fetch a real data block from the server, the client will download all blocks of the associated path and find the correct block.
- **Oblivious write**. After accessing the block of the path, the block been read will be assigned to another random leaf. Then the block is popped into the root node obliviously from the client-side to the server; meanwhile, all blocks in the path are re-encrypted.

Tree-path eviction. The data blocks should be shuffled to avoid the overflow of node size along the tree path during once eviction process. During a general tree-path eviction, the data blocks matching the evicted tree path should be evicted to the child node along the tree path.

Correctness. For any access sequence x, $\{ORAM_1(x), \cdots, ORAM_q(x)\}$ returns data consistent with x except with a negligible probability, where q is the number of access times.

Security. The security definition of ORAM ensures that the data access patterns from two sequences of read/write operations *with the same length* must be indistinguishable. However, the security of ORAM does not hide size information. i.e., the server knows how many addresses are accessed for a sequence of ORAM operations. The reason is that there are still some patterns that can be observed to deduce the size pattern. E.g., the communication per access is measurable and fixed, by the total communication consummation, the adversary can easily deduce how many addresses are accessed, and learn the size pattern information easily.

4 Challenges and Core Techniques

Challenges. Current ORAM-inspired SSE schemes (such as SPS14 [4], TWORAM [39], ORION [17]), and HORUS [17]) cannot achieve the goal of *strong FP&BP*. Specifically, though existing ORAM-inspired schemes can provide certain privacy enhancement such as hiding search and access pattern, they cannot hide size pattern naturally.

Another challenge is about efficiency. First, existing ORAM-inspired SSE schemes leverage single-server ORAM to achieve certain privacy goal, but even the most efficient PATH-ORAM, the client still needs to pay $O(\log N)$ blocks of communication overhead plus client-side computation such as decryption and encryption. Second, the definition of the data block of existing ORAM-inspired schemes also make the efficiency worse; they cannot meet the strong privacy goal here. Specifically, they usually define the data block as a pair of (w, ind): (1) the position of each data block will be changed with the oblivious read and oblivious write operations of ORAM, making the inverted index difficult to be built effectively and resulting in inefficient search operation; (2) when searching a keyword, file identifiers associated with this keyword are stored in several different data blocks, which will be obtained through multiple oblivious read operations from the server, and the number of interactions will leak the size pattern; (3) when updating a file, each keyword it contains will be written to the server as a data block, resulting in multiple interactions with the server, and the number of interactions will also leak the size pattern; (4) whatever updating a file or searching a keyword, it is accompanied by a large number of dummy block read and write operations, which incurs excessive computation and communication burden.

In summary, due to existing security and efficiency issues, designing SSE schemes with strong privacy definition, meanwhile maintaining satisfying efficiency is still a challenge.

Overview of our technique. In order to eliminate leakage of searchable encryption, the standard approach is to use ORAM to hide search and access pattern, and the main issue is efficiency. As we showed previously, using single-server ORAM can render a considerable overhead. To resolve the issue, we use a multi-server based ORAM scheme, i.e., S^{3}ORAM [40], to reduce client-server communication, which alleviates the burden of the client in the ORAM protocol meanwhile concealing search/access pattern. We also design a novel eviction protocol based on matrix permutation and Shamir secret sharing; the eviction protocol is more efficient than the one in S^{3}ORAM. Besides, we design data block by packing search result into one single data block, which is much efficient than previous schemes that define data block per keyword/identifier pair. We also define update slots to improve efficiency during the update further.

In the following, we introduce our techniques in more details.

4.1 Data Block Definition

To hide the size pattern in the search operation, we should make the search result to be the same length. For a keyword, we define a data block to store its related file identifiers. As shown in Fig 3, we define each element in a data block as a special file identifier ind. There are at most C file identifiers in a data block, where C is the maximum number of files associated with a keyword. Generally speaking, with a proper word segmentation algorithm, C is much smaller than F, where F is the maximum number of files in the databases.

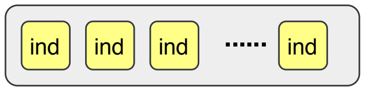


Fig. 3: Data block definition.

4.2 Update Slots and SSE Operations

When the block is defined as above, the keyword search operation will become very convenient, but the update operation will still leak the size pattern. Because a file contains multiple keywords, the update operation will modify each data block corresponding to each keyword, that is, it must inevitably interact with the server multiple times, and the size pattern will still be leaked. How to protect the size pattern in the update operation is still a challenge.

In order to hide the size pattern, when uploading a file, we must unify the number of keywords it contained to a pre-defined value M, where $M \geq K$ and K is the number of all distinct keywords. A very intuitive way is to expand the empty keywords to ensure that the total number of keywords to be uploaded reaches M. However, it will bring a lot of communication and computation overhead. Take a tree-based ORAM with the capacity of $\mathcal{N}$ as a black box, the update operation will result in M oblivious data accesses, and there will be at least $M \cdot \log \mathcal{N} \cdot F$ communication overhead. Obviously, this method is very inefficient.

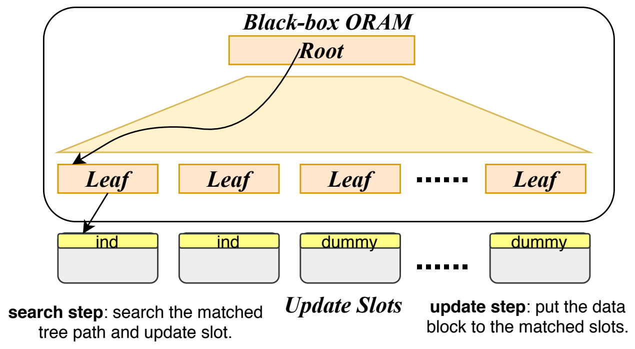

Black-box ORAM
Root
Leaf
Leaf
Leaf
......
Leaf
ind
ind
dummy
......
dummy
search step: search the matched tree path and update slot.
Update Slots
update step: put the data block to the matched slots.

Fig. 4: The server storage in Eurus.

Update slots. The update slots are prompted to solve the problem of update efficiency. As shown in Figure 4, the update slot is set for each path of the black-box ORAM, and each path is corresponding to a keyword (real or dummy). For each update slot, it stores file identifiers uploaded during update operations.

Update operation. When adding a file (the identifier is ind), a file identifier or a dummy value will be written into each slot. If the keyword corresponding to the slot is included in this file, the file identifier ind will be written; otherwise, a dummy value (for example, defined as 0) will be written. When a file is deleted, its inverse file identifier (negative ind) or a dummy value will be uploaded. In this case, no data access operation of ORAM is executed. Although all update slots are involved, the communication bandwidth is only related to the number of keywords K, because the number of paths is set to $c \cdot K$ in the black-box ORAM, where c is a constant value. Therefore, the efficiency has been greatly improved.

Search operation. There are three steps to perform a keyword search query. In the first step, the data block corresponding to the keyword is read obliviously from the path related to this keyword; in the second step, the file identifiers stored in the corresponding update slot are retrieved; in the third step, the client merges two parts of data, gets the retrieved results, then randomly assigns a new path to the merged data block and writes it back to the root node obliviously.

Eviction operation. It is a necessary step for ORAM. It obliviously moves the data block corresponding to the keyword written to the root node to a deeper position, ensuring that the root node can write more new data blocks. In Section 5.3, We combine the multi-party computation protocols and the matrix permutation technique to optimize the tree-path eviction in the black-box ORAM.

5 EURUS: OUR CONCRETE CONSTRUCTION

In this section, we propose an efficient SSE scheme named "Eurus" to achieve *strong FP&BP*. Considering the efficiency, we select S^{3}ORAM [40] as the underlying ORAM scheme, which works in a multi-server computation setting. We also further design an efficient eviction method to reduce the number of data blocks along a tree path. The main reason for using multiple servers is to improve efficiency under the assumption that the adversary is hard to corrupt servers from different service providers and different service providers are unwilling/hard to conclude with each other due to conflict of interest or existing privacy regulation. Under the assumption, some SSE schemes [50] [26] in multi-server setting was designed previously.

5.1 Storage Structure

Server storage. We take an S^{3}ORAM tree structure $\mathbf{I}$ to store data blocks that are defined in Section 4.1. The real data block stores the file identifiers associated with a keyword, and the dummy data block stores the encryption of zero. All data blocks have the same length and contain no more than F file identifiers. For each node in the tree, it contains Z blocks, where Z is a constant value (usually 5). We set the total number of leaf nodes in the tree as well as the update slots to $c \cdot K$, where c is a constant value and $c \geq 2$. Denote $\mathbf{U}$ as update slots and $\mathbf{U}_i$ as the i-th update slot, where $i \leq c \cdot K$.

The S^{3}ORAM [40] framework requires n ($n \geq 3$) servers, and each server has the same storage structure. We denote S_i as the i-th server. For a data block D, it will be divided into n shares by Shamir secret sharing, and the share D_i will be stored on the server S_i.

Client storage. We use a hash table $\mathbf{T}$ to store the keyword information, and define $\mathbf{T}$:=<*key*, *value*>, where *key* is a keyword w and *value* is the pair containing ID of the path l and the data block identifier ind, which can be retrieved as *value*←$\mathbf{T}$[key]. Meanwhile, we define $\widetilde{\mathbf{T}}[l]$ to identify the keyword w corresponding to the path l. We define an array $\mathbf{Y}$ to record whether the tree path at the server side is touched or not. In addition, the client uses a position map to store the state of data blocks. It can be stored on the server-side just like other ORAM schemes, and accessed in a recursive manner. The client also needs to store the key used for data encryption.

5.2 Building Blocks

Shamir secret sharing. The main idea of a (k, n)-Shamir secret sharing [51] is to divide a secret data D to n pieces (i.e., $D_1, \cdots, D_n$), which will be further stored on different servers. To recover the secret data D, the client must collect at least $k+1$ data pieces and compute the whole data value by *Lagrange interpolation*, where k ($k < n$) can be called as the privacy level. We assume that no more than k servers can conclude in our scheme. We denote the sharing protocol as Π_{split} and the recover protocol as Π_{recover}.

Multi-party computation protocols. Based on Shair secret sharing [51], we define two multi-party computation protocols. One is multi-party addition protocol Π_{add}, which makes two secret data can be homomorphic-added. The other is multi-party multiplication protocol Π_{mul} making two secret data can be homomorphic-multiplied, which can be done by SMP protocol [52].

Definition 1. [**Multi-party homomorphic protocol** Π_{add}]: There are two secret data D and $\mathcal{D}$, where $D_i^{(k)}$ is the Shamir share of D for S_i and ${\mathcal{D}_i}^{(k)}$ is the Shamir share of $\mathcal{D}$ for $\mathcal{S}_i$ ($1 \leq i \leq n$), we have

$${D_i}^{(k)} + {\mathcal{D}_i}^{(k)} = (D + \mathcal{D})_i^{(k)}.$$

Definition 2. [**Multi-party multiplication protocol** Π_{mul}]: There are two secret data D and $\mathcal{D}$, where $D_i^{(k)}$ is the Shamir share of D for S_i and ${\mathcal{D}_i}^{(k)}$ is the Shamir share of $\mathcal{D}$ for $\mathcal{S}_i$ ($1 \leq i \leq n$), we have

$${D_i}^{(k)} \cdot {\mathcal{D}_i}^{(k)} = (D \cdot \mathcal{D})_i^{(2k)}.$$

It can result in a share of $D \cdot \mathcal{D}$, which is shared by a random polynomial with order $2k$.
To reduce the value of polynomial, each party can perform the inner product between the received shares, and then compute $(D_i \cdot \mathcal{D}_i)^{(k)}$ by the $\star$ operator defined in [40], that is $(D \cdot \mathcal{D})_i^{(k)} = {D_i}^{(k)} \star {\mathcal{D}_i}^{(k)}$, which achieves degree reduction from $2k$ to k.

S^{3}ORAM. In S^{3}ORAM [40], data access is the same as the tree-based ORAM described in Section 3.3, but implemented by Shamir secret sharing [51] and multi-party computation protocols [52] described above. According to its definition, S^{3}ORAM.Setup() defines the initialization process, and S^{3}ORAM.retrieve(id) defines a data block oblivious read process with its identifier id, which will be used in the rest of the paper. We mainly review the eviction operation here.

The S^3ORAM [40] performs the tree-path eviction according to the reverse lexicographical order [43]. One tree-path eviction contains $O(\log \mathcal{N})$ triplet evictions and each triplet eviction involves a parent bucket and two child buckets, where $\mathcal{N}$ is the capacity of the ORAM. The parent bucket is the source, the child bucket along the eviction path is the destination. The data blocks corresponding to the path in the source will be evicted to the target child bucket. The other child bucket called sibling can be copied from the source since non-leaf sibling buckets are guaranteed to be empty by prior evictions except with a negligible bucket overflow probability. This eviction method requires that every bucket contains $O(\log \mathcal{N})$ data blocks, and once triple eviction needs to permute $O(\log^2 \mathcal{N})$ data blocks, resulting in total $O(\log^3 \mathcal{N})$ computations. It is still a heavy computation overhead for designing a new SSE scheme.

5.3 Efficient Eviction Operation Design

We try to optimize the eviction operation and prompt a *thin tree-path eviction* that only needs to permute all blocks on the path once. The core idea of this eviction operation is to generate a sequence of permutations by querying the

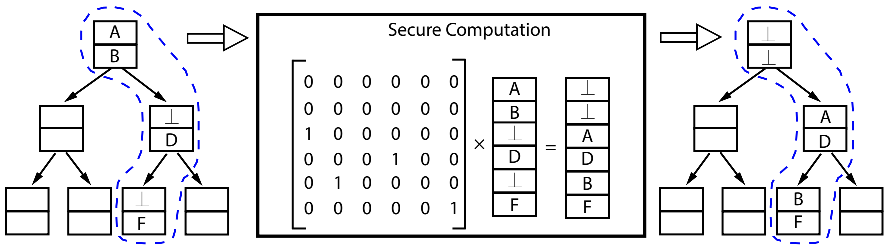


Fig. 5: **ThinEvict over the whole eviction path.** The original data blocks in the eviction path are organized as $(A, B, \perp, D, \perp, F)$, the client shares an eviction matrix to the servers, the servers jointly perform block reorganization using a secret-shared eviction matrix. In the end, the blocks are organized as $(\perp, \perp, A, D, B, F)$, shared among servers. Throughout the whole process, servers cannot see the involved blocks or eviction matrix because the computation is run by secret-shared secure computation.

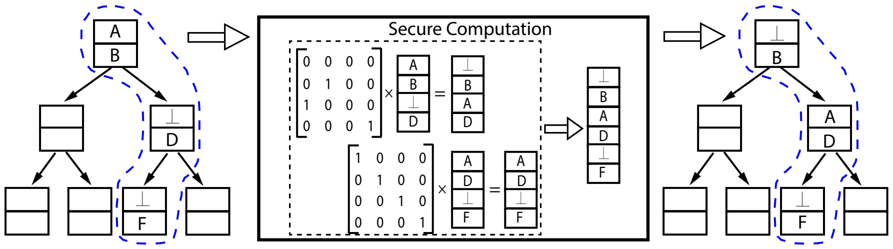


Fig. 6: **ThinEvict by a level-to-level eviction strategy.** The original data blocks in the eviction path are organized as $(A, B, \perp, D, \perp, F)$. The client shares two eviction matrix to the servers, the servers jointly perform block reorganization from the root to the leaf. The servers first jointly permutation $(A, B, \perp, D)$ to $(\perp, B, A, D)$ using the first eviction matrix, and then use the second matrix to organize $(A, D, \perp, F)$ to $(A, D, \perp, F)$. In the end, the blocks are organized as $(\perp, B, A, D, \perp, F)$.

state of the blocks on the path, and make all blocks belongs to this path to be permuted to the deepest positions as much as possible. The thin tree-path eviction adopts the eviction strategy of prior tree-based ORAM [41], [44], but we initiate it with matrix permutation algorithm to achieve permutation obliviously for all block on the evicted path. Therefore, the underlying eviction procedure can be initialized with Path-ORAM or Circuit-ORAM eviction strategy[1]. Specifically, the thin eviction strategy contains three phases:

- *Initialization phase*. We use P to denote the eviction path and $|P|$ to denote the size of eviction path.
- *Matrix generation phase*. We set all the evicted blocks into one dimensional matrix as $e = (id_1, ..., id_{|P|})$. Then, we generate an eviction matrix $\mathcal{I}$, the size of which is $|P| \times |P|$, where $\mathcal{I}[i, j] \in \{0, 1\}$. If the data block at position i before tree path eviction wants to be moved to the position j after eviction, we should set $\mathcal{I}[i, j] \leftarrow 1$. Otherwise, set $\mathcal{I}[i, j] \leftarrow 0$. Notice that the eviction matrix generation follows the "go-furthest" rule, meaning that all the data blocks in the evicted path should be evicted to the furthest bucket.
- *Multiplication phase*. The oblivious permutation can be completed by the multiplication of the above data matrix e and the eviction matrix $\mathcal{I}$. The result is a 1-dimensional matrix containing the permuted data blocks.

Thin tree-path eviction. The eviction still follows the reverse lexicographical order [43] [42] and go-furthest rule [41]. The *go-furthest* means that every data block matches a tree path, all the data blocks in the evicted path are evicted to the furthest bucket, which is the overlap of the matching tree path and the evicted tree path. Each path eviction is achieved by the above matrix permutation algorithm.

We give an example to illustrate its realization. Assume that data blocks in the eviction path P are represented as the 1-dimensional matrix $(id_1, id_2, id_3, ...)$, and we have the eviction matrix $\mathcal{I}$ as follows,

$$\mathcal{I} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Data blocks in the eviction path can be obliviously permutated after multiplication with the eviction matrix as follows:

$$\mathcal{I} \cdot (id_1, id_2, id_3, ...)^{\mathrm{T}} = (id_2, id_1, id_3, ...)^{\mathrm{T}}$$

We can see that in this eviction, the first two blocks of data are swapped after multiplication operation. We give a concrete example in Fig. 5 about how to organize the blocks on the eviction path.

For the security, the matrixes e and $\mathcal{I}$ can be hidden their information for the server-side by the multi-party protocol defined in definition 2. Therefore, the malicious server cannot track the block eviction, and the eviction operation can be done obliviously. In Eurus, the capacity of a bucket is Z

1. This means the previous analysis of ORAM characteristics in [41], [44] can be applied to thin eviction without modification

(usually 5) and only $O(\log \mathcal{N})$ data blocks are permutated during a path eviction. We define this eviction process as the function **ThinEvict** and its algorithm is shown in Figure 7.

An alternative eviction strategy with $O(\log \mathcal{N})$ communication. The idea of prior **ThinEvict** protocol is to perform permutation over all blocks on the evicted path, with $O(\log^2 \mathcal{N})$ communication cost and $O(1)$ round complexity. We can borrow the idea of Circuit-ORAM eviction strategy to enable **ThinEvict** with $O(\log \mathcal{N})$ communication complexity but in $O(\log \mathcal{N})$ rounds. By using the secret-shared matrix permutation technique, the Circuit-ORAM eviction can be performed by $O(\log \mathcal{N})$ permutations, each is being over $2Z + 1$ elements (see [41] for more details), which results in an $O(\log \mathcal{N})$ communication complexity. However, these permutations must be performed one-by-one from the root to the leaf, incurring $O(\log \mathcal{N})$ round complexity. The trade-off between communication and round complexity provides certain flexibility, which trade-off is more preferred depends on the network condition. E.g., eviction with less round but higher communication complexity can be more suitable when server-server network is in good condition.

We give the concrete example in Fig. 6 on how to organize blocks by a level-to-level eviction strategy.

5.4 Concrete SSE Algorithms

As Figure 7 shows, the process of SSE basic operation consists of *setup*, *search*, *update*, and *thin tree-path eviction*.

Setup. The client generates a secret key key$_{prf}$, initializes the hash table $\mathbf{T}$ and the empty array $\mathbf{Y}$, where $\mathbf{Y}$ records whether a path is searched or not. The server initializes the index tree $\mathbf{I}$. All the data blocks in $\mathbf{I}$ are randomly generated which can be used as dummy blocks in the future and each data block can be split into n secret shares for each server. For oblivious eviction, the server initializes the eviction counter cnt and cnt', the former is for normal ORAM eviction, and the latter is for update slot eviction described in Section 5.5.

Update. The update process is executed in the update slots. As described in Section 4.2, when a file is added to the server, its identifier ind is written to the corresponding update slots while the dummy zeroed data is put into the other update slots. When the file is deleted from the server-side, the inverse of ind should be written. The entire execution process is completed by secret sharing and secure multi-party computation.

After successive update operations, the update slots may be full and cannot store new file identifiers. Although search operation will empty update slot, we consider the possibility of continuous update operations. In this case, update slot eviction is required to empty update slot. The real update operation related to a keyword is executed at the client-side, when file identifiers are downloaded to client during search operation or update slot eviction.

Search. The search operation includes four steps: Step 1 is retrieving the data block related to the keyword w through the black-box ORAM's oblivious read operation; step 2 is reading all the updated file identifiers in the update slot and empty this update slot; step 3 is merging the file identifiers in the data block and update slots at the client side, and step 4 is writing the merged data block back to servers and executing thin tree-path eviction. The new merged data block is matched to an unused tree path selected by a Pseudo-Random Function **F**. The search tokens are the path l, and the data block id related to the keyword w. By taking (l, id) as input, the function **TwoPartRetrieve**(l, id) can be done the search operation based on the black-box ORAM and update slots. Finally, the client will set $\mathbf{Y}[l] = 1$ to denote the update slot has been emptied.

Although S^3 ORAM [40] is used as the block-box ORAM and the search operation can be directly completed through its data access function, we have modified its eviction operation for the consideration of efficiency. This is why our pseudo code does not use S^{3}ORAM's access function but divides it into S^{3}ORAM.retrieve(id) and our defined function **ThinEvict**. The single difference is that the S^{3}ORAM [40] should select a data block from all the nodes along the tree path to execute the retrieve function; however, our scheme needs all the data blocks in all the nodes along the tree path to take part in the retrieve function. The computation cost and bandwidth cost in our methods are $O(\log^2 K)$.

5.5 Additional Operations

Dynamically increase data block size. Because the size of the data block is associated with the maximum number of files associated with a keyword, considering efficiency, we allow the SSE scheme to adjust the size of C dynamically. Initially, a relatively small C can be set by the client. Once the current value of C cannot meet the requirements, Eurus can dynamically increase the value of C by padding secret shares of zero to the data block to increase the block size. This operation does not affect the eviction operation and is easy to perform without revealing the size pattern.

Evict the blocks in the update slot. After several update process, some update slots might be full. To avoid this, we call function **UpdateEvict**() to evict file identifiers in update slot to its data block, we call this as "update slot eviction". It works like the normal ORAM eviction. For the consideration of security, merging file identifiers into its related data block should be finished at the client-side. Therefore, we achieve this goal by executing fake keyword search and the fake keyword, which is selected by the reverse lexicographical order same as the normal ORAM eviction strategy. If the update slot has been touched by a search query, the chosen eviction for it will be terminated. Therefore, update eviction operation will not be happened frequently when search often executes.

6 Analysis

6.1 Capacity Analysis

We can draw the conclusion that the size of server-side is controllable and acceptable, by the design of black-box ORAM and update slots. The quantity of update slots is $c{\cdot}K$, where c is a constant value. The capacity of each update slot is dependent on the frequency of update queries, search queries, and UpdateEvict function. Each update slot stores the updated file identifier ind, which can be controlled to a small capacity by downloading the ind to the client and then uploading it to the black-box ORAM during a keyword

```
Setup(λ, ⊥) :
Client:
1:  key_prf ← {0,1}^λ
2:  T ← empty hash table
3:  Y ← empty array, //record whether the path is searched or not.
4:  {I_i}_{i=1}^n ← S^3ORAM.Setup()
5:  Send I_i to the server S_i
 Server {S_i}_{i=1}^n :
6:  I ← I_i, cnt, cnt' ← 0 //cnt,cnt' is a global counter.
Update(σ, op, ind) :
Client:
1:  For each keyword w ∈ file ind do
2:      If T[w] is null then
3:          T[w].l ← a random unused path
4:          T[w].id ← new data block identifier
5:  B ← empty array
6:  If op is delete then
7:      For i = 0 to c · K do
8:          B[i]← −ind
9:  else
10:     For i = 0 to c · K do
11:         B[i]← 0, w ← T̃[i]
12:         If w is in file ind then B[i]← ind
13:  {B_i}_{i=1}^n ← Π_split(B)
14:  Send B_i to the server S_i
15:  Execute UpdateEvict.
 Server {S_i}_{i=1}^n :
16:  B* ← B_i                // the received array
17:  For j = 0 to c · K do
18:      Add B*[j] to update slot U_j
UpdateEvict():
Client:
1:  cnt' ← cnt' + 1 mod(c · K)  //add 1 to the global counter.
2:  i ← cnt', w ← T̃[i], (l, id) ← T[w]
3:  If Y[i] == 0 then TwoPartRetrieve(l, id)
4:  If cnt' == 0 then Y ← empty array
  //all tree paths are touched more than once, initialize the K.

Search(w):
Client:
1:  (l, id) ← T[w]      //Get the path and the data block id
2:  Y[l] = 1  //record the path l has been touched
3:  Execute TwoPartRetrieve(l, id)

TwoPartRetrieve(l, id):
 Server {S_i}_{i=1}^n :
1:  B_i ← U_l //collet file identifers in update slot, clear U_l.
2:  Send B_i to the client
Client:
3:  D ← S^3ORAM.Retrieve(id)
4:  //all the data blocks in the l needs to retireve.
5:  B ← Π_recover({B_i}_{i=1}^n)
6:  D* ← Merge(D,B)
7:  // Merge file identifiers in array B into block D
8:  T[w].l ← a new unused path chosen by the function F.
9:  Write D* to the root of black-box ORAM obliviously.
10:   Execute the ThinEvict.

ThinEvict():
Client:
1:  cnt ← cnt + 1 mod(c · K) //add 1 to the global counter.
2:  l ← cnt
3:  |P| ← the number of blocks stored along the path P_l.
4:  I ← zeroed |P| × |P| eviction matrix.
5:  e ← (id_1, ··· , id_|P|) //the evicted data blocks along P_l.
6:  For t = 1 to |P|, do
7:      k ← gofurthest[id_t];
8:      I[t, k] = 1.
9:  //gofurthest makes blocks evicted to the k-th location in e.
10:  {I_i}_{i=1}^n ← Π_split(I)
11:  Send I_i to the server S_i
 Server {S_i}_{i=1}^n :
12:  e_i ← the secret share of e in server S_i
13:  e_i* ← Π_mul(e_i, I_i)     //multiplication protocol
```

Fig. 7: The algorithms of Eurus scheme.

search process. The UpdateEvict can be done after every update process. The maximum capacity of the update slot is no more than $c \cdot K$ data blocks.

The black-box ORAM adopts the algorithm of go-furthest tree-path eviction in Circuit ORAM [41] but added into the multi-server environment to improve its communication and computation overhead. The Circuit ORAM [41] has been proved that the capacity for a node in the binary tree can be set to 5. The node size in the black-box ORAM can be controlled to a constant value of Z, which can be proved in Theorem 1, which refers to [41].

Lemma 1. Let R denote the number of blocks overflowing from the binary ORAM tree. Let $u^T(\textit{black-boxORAM}[s])$ denote the number of blocks contained in the nodes of the subtree T after the eviction sequence s. The T is a subtree with $n = n(T)$ nodes containing the root of the binary ORAM tree. Then, for $Z > 5$, for any $R > 0$,

$$\Pr[u^T(\textit{black-boxORAM}[s]) > n{\cdot}Z{+}R] \le \frac{3 \cdot (0.93312)^n}{4^n} {\cdot} (0.6)^{-R}$$

Theorem 1. Let st($\textit{black-boxORAM}^Z[s]$) be a random variable denoting the cache size after eviction sequence s for the Eurus with the node size $Z > 5$. Then, we have,

$$\Pr[st(\textit{black-boxORAM}^Z[s]) > R] \le 42 \cdot 0.6^R,$$

where the probability is taken over the algorithm's randomness.

Proof 1. Based on Lemma 1 and the inequality as follows, $\Pr[st(\textit{black-boxORAM}^Z[s]) > R] \le \sum_{n \ge 1} 4^n \max_{T:n(T)=n} \Pr[u^T(\textit{black-box ORAM}[s]) > nZ + R]$. We can get the result:

$$\Pr[st(\textit{black-boxORAM}^Z[s]) > R] \le 42 \cdot 0.6^R.$$

Therefore, Z is up to 5 is enough to complete the operations in black-box ORAM.

6.2 Security Analysis

The security can be proved by the form like in TWORAM [39]. All the operation security analysis is based on the black-box ORAM and the update slots.

Theorem 2. The black-box ORAM is correct. For any query sequence x, $\{ORAM_1(x), \cdots, ORAM_q(x)\}$ returns data consistent with x except with the division of data retrieve, write, and eviction, where q is the number of queries to the ORAM.

Proof 2 Let $ORAM_i(x)$ be the Eurus client's sequence of interactions with server S_i including a sequence of search, update, and eviction operations to the black-box ORAM. Due to the independence and different execution process among the above operations, we can consider $ORAM_i(x)$ to contain the separate sequences of these operations observed by S_i as follows.

Search transcripts: For each keyword search request $y_i \in x$, S_i observes that once a data block in the black-box ORAM or the update slots is accessed, its position is assigned to a new bucket leaf or a new update slot selected which looks random. The retrieval process referred to S^3ORAM has been proved secure by using PIR technique. The corresponding retrieval transcripts in the black-box ORAM are information-theoretically indistinguishable from a random sequence.

Update transcripts: For each update request $z_i \in x$, S_i observes that all the update slots have been padded with a real file identifier or a dummy data block. The position of the real data block has been hidden simply. All the update processes can look the same for the adversary.

The UpdateEvict is done by a constant selection rule. The main process is to search a keyword, which is proved previously. The UpdateEvict can make the update slot have enough capacity to support the frequent update operations. The frequency of the same update slot search queries during the keyword search query can leak no information for that the search position in the update slots looks random for the adversary.

ThinEvict transcripts: The ThinEvict consists of several multi-party multiplications between the data blocks and the eviction matrixes. Let $m = \{\prod_{mul_1}, ..., \prod_{\mathrm{mul_l}}\}$, $k = \{\prod_{\mathrm{mul}_{1'}}, ..., \prod_{\mathrm{mul}_{l'}}\}$, $Q \subseteq \{1, ..., l\}$. The eviction transcripts $\{E_{i \in Q}(m)\}$ and $\{E_{i \in Q}\ (k)\}$ observed by the honest but curious servers are computationally indistinguishable. The data generated in independent evictions through m and k are identically distributed. Since the eviction process is deterministic by the design of eviction matrix $\mathcal{I}$. The client is trusted and products the eviction matrix $\mathcal{I}$.

For each eviction, the matrix should be split to n shares for n servers. They are uniformly distributed. All the servers cannot be corrupted. Therefore, all the servers cannot know the whole information from the eviction matrix $\mathcal{I}$. Therefore, the result from the multiplication protocol is hidden by the MPC protocol. Therefore, the eviction can be achieved obliviously.

Theorem 1. The Eurus is strong forward and backward private, if black-box ORAM is secure, F is a PRF, and the encryption is IND-CPA-secure.

Proof 3. We first describe a simulator Sim who generates the transcripts for the ideal distribution $SSE_{ideal\,S,\mathcal{A},\mathcal{L}}(\lambda)$. To generate the full transcripts of the back-box ORAM scheme for the adversary $\mathcal{A}$, The simulator Sim needs to simulate the scheme due to the security in Theorem 2. After every search query, the corresponding tree path for the search result should be selected by a PRF. In the Sim, it replaces the PRF-generated paths by the uniformly random paths. The encryption can be the ciphertexts of 0 simply. Sim knows the number of files corresponding to all the keywords. We need to show that $SSE_{ideal\,S,\mathcal{A},\mathcal{L}}(\lambda)$ is indistinguishable from $SSE_{real}{}^{\Pi}_{\mathcal{A}}(\lambda)$.

The proof follows by a hybrid argument.

H_0: The hybrid matches the real execution. $\mathcal{A}$ chooses DB. After the setup process, $\mathcal{A}$ adaptively execute *Search*(w) and then denote the full transcripts of the protocol by t_i. The update query can be done in the same way. Finally, (EDB, $t_1, \cdots, t_q$) is as an output, where q is the number of queries by the $\mathcal{A}$.

H_1: Based on the H_0, during the search process, the retrieve, write, and eviction operations based on the black-box ORAM are proved by the experiment in the Theorem 2.

The indistinguishability of H_0 and H_1 is based on the security of ORAM scheme.

H_2: Based on the H_1, the only difference is that the ciphertexts are replaced by the encryption of 0. The length of every data block in the black-box ORAM and update slots is leaked by the leakage function $\mathcal{L}$ ($\mathcal{L}_{Search}(w) = (\mathcal{F})$), which depends on $\mathcal{F}$, that is, the number of files. The update function is done in the update slots, replaced by the encryption of 0 addition and eviction to the black-box ORAM. The only leakage function is that $\mathcal{L}_{Update}(\sigma, op, ind) = (UpdateTime(W))$.

The indistinguishability of them is based on the encryption scheme used in the black-box ORAM and update slots.

H_3: Based on the H_2, the only difference is that the PRF-generated tree paths are replaced by the uniformly random tree paths. The H_3 complete all the ideal-state experiment.

7 Implementation and Evaluation

We care about the performance of SSE operations and evaluate them from three aspects: (1) response time and bandwidth cost of a keyword search query; (2) the performance of update query; (3) execution time of the tree-path eviction operation.

7.1 Experimental Details

Target SSE schemes. In order to accurately evaluate performance, we chose two SSE schemes:

- **SPS14** [4]. The first SSE scheme achieves forward privacy, which is constructed based on layer-based ORAM. However, it cannot protect the size pattern and protect the access pattern.
- **GOSSE**. We tried to build an ORAM-inspired SSE solution achieving strong FP&BP, which is more secure than TWORAM [39] and ORION [17], and named it as "GOSSE". Like TWORAM and ORION, GOSSE also defines the data block in the index structure as (w, ind) pair, which presents the relationship of the keyword and the file. Meanwhile, it achieves strong FP&BP by padding some dummy data block operations in the update and search operations. Most important of all, it adopts the fastest ORAM model

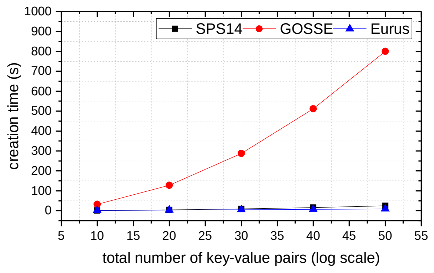


Fig. 8: The database creation for different data block design.

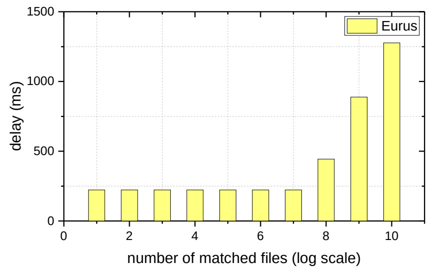


Fig. 9: The response time of data block merge in Eurus.

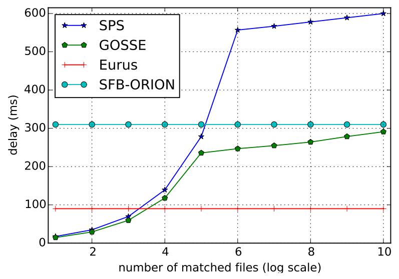


Fig. 10: The execution time for the search process, where the total number of keyword-file pairs is 2^{20}.

(S^{3}ORAM, the same as that in Eurus) to achieve the base performance. To make the efficiency comparison be fair, (w, ind) pair can be written and read obliviously in GOSSE according to the description in Section 5.2.

- **SFB-ORION**. We improve ORION [17] to be with strong FB & BP, for which we call as SFB-ORION. Note that ORION already hides search/access pattern on index-level, so we improve ORION by hiding the size pattern to obtain strong FP and BP (on index-level). This is done by performing additional dummy ORION access to make result length uniform across all search queries.

Implementation details. The GOSSE and Eurus are in a multi-server environment, and they both adopt Shamir secret sharing [51] to complete the multi-party computation. The number of servers is 5 in our experiment. The SPS14 is in the single-server environment. We choose RocksDB as server storage.

For cryptographic algorithms, AES and Blake2b are selected as the symmetric encryption algorithm and underlying hash function, respectively. The encryption key of AES is 256 bits. We use *sse* crypto library used in $\Sigma o\phi o\varsigma$ [13] as the cryptographic tools. The p for $\mathbb{F}_p$ is a 256-bit prime.

Experimental environment. All the experiments were performed on five computers with a single Intel Core i7-7700 3.6GHz CPU, 32GB of DDR3 RAM, 512GB SSD running Linux Ubuntu 14.04 LTS operating system, in the local network environment.

7.2 Dataset and Keyword Distributions

We use *Enron email* as small test dataset and use *Wikipedia* dumps as large test dataset. In *Enron email*, the file number is about 517,000 and key number is about 20,000. We adopt the *wikipedia-20150602* as the concrete large dataset, whose file number is 5,078,000, number of keyword is 70,000, and keyword/identifier number is 2^{20}.

Dataset preprocess. We make use of the NLTK library to exclude the stopwords and punctuation marks from the original database. Then we make use of PorterStemmer provided by the NLTK to extract keywords and exclude duplicate keywords in every file.

Keyword distribution. Based on the Enorn email, most of the keywords match only one file. We can conclude that the length of a data block in the black-box ORAM can be controlled largely smaller than F with a proper word segmentation, where F is the number of files, which can be described in Fig. 14. In particular, most keywords match less than 100 documents.

7.3 Experimental Results

Data block size adjustment. The element of a data block in SPS14 [4] is consist of a keyword, operation, and file identifier, where the data block size is 0.5KB. The one in GOSSE can include the keyword and file identifiers, where the data block size is 0.16KB. A data block size in the above two concrete schemes is constant. The total number of keyword-file pairs is 2^{10}. The Eurus just stores the file identifiers, each of which has the size of 0.06KB. Our Eurus scheme can adjust the data block size dynamically. The Eurus contains the extreme operation, that is, data block size adjustment. The chunk of a data block in the Eurus is 7.68KB, which can contain at most 128 file identifiers, that is to say, $\mathcal{C} = 7.48$KB. The several chunks can be collected to a whole data block at the client-side.

In the Eurus, the database created due to different schemes can be tested as Fig. 8 describes. The database is created by the S^{3}ORAM setup algorithm, and all future updates are performed by Update protocol as specified in Fig. 7. The database creation needs about 30$\times$ more response time in the GOSSE than that in SPS14 [4]. According to Fig. 9, this data block size adjustment, for example, the data block size C is from $\mathcal{C}$ to $2\mathcal{C}$ can be done suitably and take less time to complete. The delay can keep the growth rate no more than $0.6\times$ from $k \cdot \mathcal{C}$ to $(k+1) \cdot \mathcal{C}$ (k is a small constant value). Most of the data block size adjustment is made at the trusted client-side.

Keyword search efficiency. The GOSSE and Eurus make use of PIR technique [53] to fetch the data blocks related to a special keyword. The PIR technique is based on the asymmetric cryptographic algorithm, which has longer execution time. The total time of server computation is $5\%-7.8\%$ in response to the search process. In the Eurus, only a data block should be downloaded during a keyword search, rather than several data blocks. Therefore, the Eurus needs fewer communications than other SSE schemes, which causes fewer response time during the keyword search process. GOSSE, SPS14 and SFB-ORION need one more communication times than that in Eurus. According to Fig. 10, on

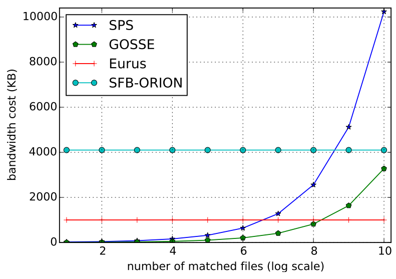


Fig. 11: The bandwidth cost for the search process, where the total number of keyword-file pairs is 2^{20}.

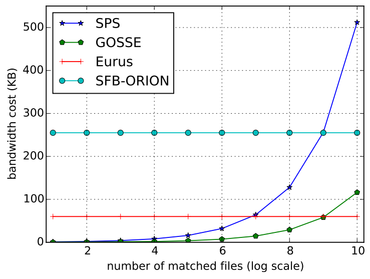


Fig. 12: The bandwidth cost of a update operation, where the total number of keywords is 2^{10}.

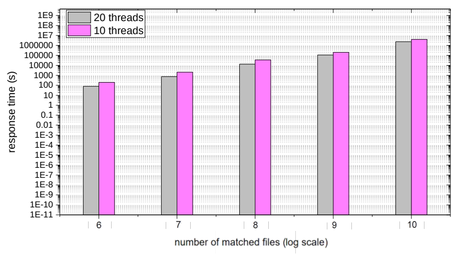


Fig. 13: The response time of Eurus in large data set.

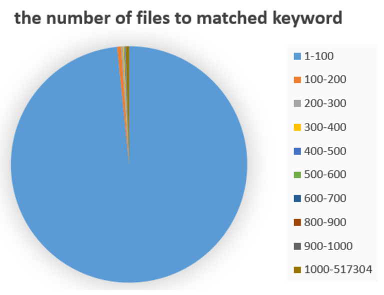


Fig. 14: The division of the number of files matched to a keyword.

TABLE 2: Comparison of operation performance during the search and update operations. (unit: the number of matched files per sec)

test times	Search		Update	
	SPS14	our scheme	our scheme	GOSSE
1	750	35740	172668	46867
2	682	36540	161435	44174
3	743	35344	232788	64278
4	800	36450	212527	57373
5	754	36193	228534	64643

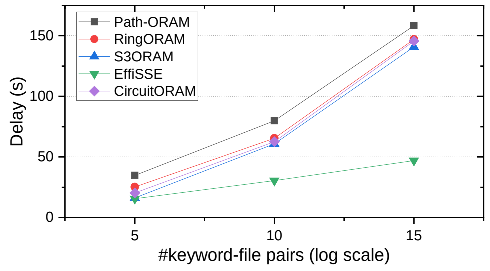


Fig. 15: The execution time for the tree-path eviction.

average, the response time of the search process in Eurus is 30% of that in GOSSE, 26% of that in SFB-ORION, and 17% of that in SPS14. The bandwidth cost in Eurus is 10% of that in SPS14, 49% of that in GOSSE, and 15% of that in SFB-ORION, which is described in Fig. 11. According to Table 2, SPS14 [4] is worse than our scheme for the response time of a search operation. The update slot eviction operation is similar to a keyword search operation, which is also with practical efficiency as we will show later.

Update efficiency. The SPS14 [4] uploads a given number of data blocks, which is equal to the number of matched files to the given keyword. The GOSSE needs to upload the same number of data blocks. The Eurus needs to upload the given number of data blocks, which is equal to the number of keywords. Therefore, the size pattern for the update process is hidden naturally. The bandwidth cost in SPS14 is $2.9\times - 4.4\times$ than that in GOSSE, and $1.5\times - 2\times$ than that in SFB-ORION. Eurus has the smallest bandwidth cost in the general update process of the these SSE schemes, which is described in Fig. 12.

During the update process, the number of matched files can be dealt $4.73\times$ faster in Eurus than that of GOSSE per second, as presented in Table 2.

Tree-path oblivious eviction efficiency. The tree-path eviction in Eurus is based on the go-deepest tree-path eviction in Circuit ORAM [41]. The path-ORAM [44] adopts the traditional eviction method, which is the eviction among the parent node and two children nodes, to reorganize all blocks on the evicted path. The Ring ORAM [43] is based on the tree-path eviction, which has better performance than the path-ORAM. The S^3ORAM makes the improvements on the tree-path eviction of Ring ORAM [43]. Fig. 15 shows that the execution time of tree-path eviction in Eurus is $2.34\times$ faster than S^3ORAM, $2.56\times$ faster than RingORAM, $2.94\times$ faster than PathORAM. Our eviction requires server-server interaction. We calculate overhead of the interaction; it takes around 6% of the total computation time.

Evaluation in large data set. To prove the feasibility of Eurus over the large data set, we compared the response times of search and update operations on a large dataset. We adopt the Wikipedia-20150602 as the large dataset. We perform the experiment on the dataset with result size ranging from 2^6 to 2^{10}. The experiment is performed with two different thread numbers, 10 and 20, respectively, the threads is used to perform computation task in parallel, e.g., the matrix computation during the eviction. We report the performance in Fig 11. For a keyword with a search result of size n, the response time counts the total time of performing updates and evictions to make the result size to n, plus the time of a final search operation to that keyword. As we can see, the response time is increased linearly with the number of matched files. With more threads, the response time is reduced, but increasing threads does not impact significantly on the overall efficiency.

8 Conclusions

In this paper, we proposed a DSSE scheme, named Eurus, which can hide search/access/size pattern, as well as the linkage among queries, efficiently. Technically, the techniques used in Eurus are interesting and useful. The only limitation of Eurus is that we are not able to achieve the efficiency of DSSE scheme without the same goal of security. It would be interesting to see that if an attack can be launched based on this leakage or an equally-efficient scheme can be designed also to hide these patterns. Our proposed solution is currently a practical and secure solution for DSSE for outsourcing data and search to a third-party cloud provider.

Acknowledgment

The authors would like to thank the editor and the anonymous reviewers for their help in improving the paper. This work is supported by the National Natural Science Foundation of China (No. 62032012), National Key Research and Development Program of China (No. 2020YFB1005700) and CNKLSTISS.

References

[1] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 591, 2016.

[2] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *IEEE (SP).*, 2000, pp. 44–55.

[3] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 310–320.

[4] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage." in *NDSS*, 2014.

[5] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[6] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation." in *NDSS*, 2012.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *ACM CCS*, 2015, pp. 668–679.

[8] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security*, 2016, pp. 707–720.

[9] J. Ning, J. Xu, K. Liang, F. Zhang, and E.-C. Chang, "Passive attacks against searchable encryption," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 789–802, 2018.

[10] 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*, 2016, pp. 1341–1352.

[11] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution." *IACR Cryptology ePrint Archive*, p. 441, 2019.

[12] E. A. Markatou and R. Tamassia, "Full database reconstruction with access and search pattern leakage." *IACR Cryptology ePrint Archive*, p. 395, 2019.

[13] R. Bost, "$\Sigma o\varphi o\varsigma$: Forward secure searchable encryption," in *ACM CCS*, 2016, pp. 1143–1154.

[14] R. W. F. Lai and S. S. M. Chow, "Forward-secure searchable encryption on labeled bipartite graphs," in *ACNS*, 2017, pp. 478–497.

[15] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, "Efficient dynamic searchable encryption with forward privacy," *PoPETs*, vol. 2018, no. 1, pp. 5–20, 2018.

[16] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *ACM CCS*, 2017, pp. 1465–1482.

[17] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *ACM CCS*. ACM, 2018, pp. 1038–1055.

[18] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized i/o efficiency," *IEEE TDSC*, 2018.

[19] J. Li, Y. Huang, Y. Wei, S. Lv, Z. Liu, C. Dong, and W. Lou, "Searchable symmetric encryption with forward search privacy," *IEEE Transactions on Dependable and Secure Computing*, 2019.

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

[21] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1175, 2019.

[22] G. Kellaris, G. Kollios, K. Nissim, and A. O'neill, "Generic attacks on secure outsourced databases," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1329–1340.

[23] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 315–331.

[24] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted databases: New volume attacks against range queries," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 361–378.

[25] R. Poddar, S. Wang, J. Lu, and R. A. Popa, "Practical volume-based attacks on encrypted databases," *arXiv preprint arXiv:2008.06627*, 2020.

[26] T. Hoang, A. A. Yavuz, F. B. Durak, and J. Guajardo, "Oblivious dynamic searchable encryption on distributed cloud systems," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2018, pp. 113–130.

[27] ——, "A multi-server oblivious dynamic searchable encryption framework," *Journal of Computer Security*, vol. 27, no. 6, pp. 649–676, 2019.

[28] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[29] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx." in *NDSS*, 2018.

[30] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported oram in effect: Practical oblivious search and update on very large dataset," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 172–191, 2019.

[31] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *IEEE SP*. IEEE, 2018, pp. 279–296.

[32] 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*. Springer, 2017, pp. 386–408.

[33] S. Cui, S. Belguith, M. Zhang, M. R. Asghar, and G. Russello, "Preserving access pattern privacy in sgx-assisted encrypted search," in *ICCCN*. IEEE, 2018, pp. 1–9.

[34] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx." *IACR Cryptology ePrint Archive*, vol. 2017, p. 549, 2017.

[35] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure:{SGX} cache attacks are practical," in {*USENIX*} ({*WOOT*}), 2017.

[36] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017, p. 2.

[37] 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*, 2017, pp. 523–539.

[38] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *ACM CCS*, 2017, pp. 1449–1463.

[39] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption," in *CRYPTO*, 2016, pp. 563–592.

[40] 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*, 2017, pp. 491–505.

[41] 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*, 2015, pp. 850–861.

[42] 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*. Springer, 2016, pp. 145–174.

[43] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious {RAM}," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 415–430.

[44] 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 & communications security*, 2013, pp. 299–310.

[45] M. Naveed, "The fallacy of composition of oblivious ram and searchable encryption." *IACR Cryptology ePrint Archive*, vol. 2015, p. 668, 2015.

[46] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[47] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with O((logN)^3) worst-case cost." in *ASIACRYPT*, 2011, pp. 197–214.

[48] J. Dautrich, E. Stefanov, and E. Shi, "Burst {ORAM}: Minimizing {ORAM} response times for bursty access patterns," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 749–764.

[49] 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*, 2015, pp. 837–849.

[50] T. Hoang, A. A. Yavuz, and J. Guajardo, "Practical and secure dynamic searchable encryption via oblivious access on distributed data structure," in *ACSAC*, 2016, pp. 302–313.

[51] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[52] R. Gennaro, M. O. Rabin, and T. Rabin, "Simplified vss and fast-track multiparty computations with applications to threshold cryptography," in *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, 1998, pp. 101–111.

[53] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 1995, pp. 41–50.

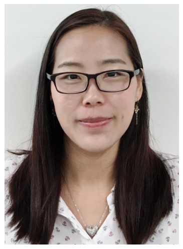
Zheli Liu received the BSc and MSc degrees in computer science from Jilin University, China, in 2002 and 2005, respectively. He received the PhD degree in computer application from Jilin University in 2009. After a postdoctoral fellowship in Nankai University, he joined the College of Computer and Control Engineering of Nankai University in 2011. Currently, he works at Nankai University as a Professor. His current research interests include applied cryptography and data privacy protection.

Yanyu Huang received Bachelor Degree of Information Security from China University of Geosciences, Wuhan, China, in 2016. Currently, she studies for the doctor degree in computer science at Nankai University. Her research interests include applied cryptography, data privacy protection.

Xiangfu Song is a Ph.D. candidate in Shandong University. He received his B.Tech in Harbin Engineering University in 2015. His research interests include applied cryptography and secure computation.

Bo Li received the BSc and MSc degrees in computer science from Nankai University, in 2017 and 2020, respectively. Her research interests include applied cryptography and data privacy protection.

Jin Li received the BS degree inmathematics from Southwest, University, in 2002 and the PhD degree in information security from Sun Yatsen University, in 2007. Currently, he works at Guangzhou University as a Professor. He has been selected as one of science and technology new star in Guangdong province. His research interests include applied cryptography and security in cloud computing. He has published over 50 research papers in refereed international conferences and journals and has served as the program chair or program committee member in many international conferences.



Yali Yuan received the M.Sc. degree from University of Lanzhou, Lanzhou, China, in 2015 and the Ph.D. degree in University of Göttingen, Göttingen, Germany in 2018 where she is currently working as a Postdoctoral Fellow. Her research interests include various topics related to wireless networks, in particular for the intelligent network and security.

Changyu Dong received the Ph.D. degree from Imperial College London. He is currently a Senior Lecturer with the School of Computing, Newcastle University. He has authored over 30 publications in international journals and conferences. His research interests include applied cryptography, trust management, data privacy, and security policies. His recent work focuses mostly on designing practical secure computation protocols. The application domains include secure cloud computing and privacy preserving data mining.