LL-ORAM: A Forward and Backward Private Oblivious RAM

Zhiqiang Wu, Xiaoyong Tang, Jin Wang, and Tan Deng

Abstract—Oblivious RAM (ORAM) enables a user to read/write her outsourced cloud data without access-pattern leakage. Not all users want a fully functional ORAM all the time since it always creates inefficiency. We show that forward-private/backward-private (FP/BP) ORAMs are also good alternatives for reducing the search-pattern leakage of dynamic searchable encryption (DSE). We introduce the FP/BP-ORAM definitions and present LL-ORAM, the first FP/BP-ORAM that achieves near-zero client storage, single-round-trip read/write, worst-case sublinear search time, and an extremely simple implementation. LL-ORAM consists of a set of switchable protocols whose security can be switched among forward privacy, backward privacy, and perfect security at any time. The construction involves a novel tree data structure named LL-tree, whose advantage is that it supports fast computation in the cloud with an access-pattern-reduced leakage profile. LL-ORAM security is formally proven under forward and backward privacy. The experimental results demonstrate that LL-ORAM is efficient and can be practically employed by DSE applications.

Index Terms—Cloud Computing, Dynamic Searchable Encryption, Oblivious RAM, Random Oracle

1 Introduction

1.1 Background and Motivation

For infinite computing and storage resources, most companies outsource their private data to the cloud. However, the cloud is perhaps insecure since many hackers can exploit system vulnerabilities or backdoors to break into the cloud systems and obtain considerable private data. Data contents are easy to be protected by using standard randomized encryption. To protect data-query privacy, the user generally relies on two modern techniques, dynamic searchable encryption, and oblivious RAM.

Dynamic searchable encryption (DSE) allows the user to store her encrypted documents on the untrusted cloud with efficient search and update capability. Assuming the user is trusted and the cloud is untrusted, all data outsourced to the cloud should remain secret. When the user wants to retrieve a document containing a specific keyword, the user only needs to send an encrypted token related to the searched keyword. Then, a set of matched documents are returned. When the user wants to insert, modify, or delete a keyword-identifier pair from the documents, the user also sends an encrypted update token. Most of the recent studies show that a DSE scheme should achieve at least forward-update privacy, which guarantees that the insertion/modification is strong, or backward-update privacy, which ensures that the deletion is also robust [1]. Otherwise, the scheme is vulnerable to hackers who can perform some attacks, such as the powerful nonadaptive and adaptive file-injection attacks [2].

- *Zhiqiang Wu, Xiaoyong Tang, Jin Wang, and Tan Deng are with the School of Computer and Communication Engineering, Changsha University of Science and Technology, Hunan, China, 410114. (e-mails: wzq@csust.edu.cn, tang_313@163.com, jinwang@csust.edu.cn,19880510@qq.com).*

Corresponding author: Jin Wang (jinwang@csust.edu.cn)

Oblivious RAM (ORAM) is a cryptographic primitive for performing oblivious read and write operations on outsourced data through reencrypting each data item and confusing the data storage location in every access. From a logical view, an ORAM can be viewed as a DSE scheme to some extent, where each document consists of only one unique keyword. Assuming security is considered, ORAMs protect the full data-query privacy, but most DSE schemes trade-off some security for efficiency. From the above analysis, it can be concluded that an ORAM still has forward and backward privacy.

Informally, a forward-private ORAM (FP-ORAM) guarantees that the write operation reveals nothing to the cloud, but the read operation can reveal the access pattern. A backward-private ORAM (BP-ORAM) guarantees that its read is oblivious, but its write can be nonoblivious. Since at least one of the operations, the read or the write reveals no access patterns, the FB/BP-ORAM is still strong enough. Our target is to study and set up the FP/BP-ORAM that can be employed by DSE schemes for enhancing DSE security.

1.2 The Need for Forward and Backward Private ORAMs

The need for building a forward-private or a backward-private ORAM comes from two aspects, efficiency and security.

Since ORAMs always create many defects, such as large client-side storage, highly interactive protocols, or heavy computational overhead, not all users like an ORAM all the time. In a write-intensive application, the user is concerned more with write access patterns, such as monitoring systems, where data are frequently written but infrequently read. A forward-private ORAM satisfies the user's requirements. In a read-intensive application, the user concerns more read access patterns, such as network disks, where the documents are rarely updated after the user uploads the

data. A backward-private ORAM can satisfy this requirement.

FP(BP)-ORAMs can improve the security of DSE schemes. The search pattern is fully revealed on most DSE applications even if they have achieved weak forward search privacy in [3]. Type-I DSE backward privacy proposed in [1] still allows a full search pattern. Note that the search-pattern leakage is the core issue of DSE since the same keyword always leads to the same search token. Adopting an FP(BP)-ORAM will contribute to addressing this issue.

1.3 Proposed Approach

We design an FP(BP)-ORAM that achieves single-round-trip read/write efficiency and near-zero client-side storage with the following approaches.

Figure 1 presents an overview of the scheme. The cloud initially stores L encrypted binary trees called LL-trees. Given the address a, to read $A[a]$, the user and the cloud use the following steps. 1) The user encrypts a into a token $Token$ and sends it to the cloud (Step A). 2) The cloud sequentially accesses Tree 1, Tree 2, $\cdots$, and Tree L (Steps B, C, D). Unlike a recursive Path ORAM [4], [5], the input of the i-th tree directly comes from the computational result of the $(i-1)$-th tree. Every input does not need to be sent back to the user for decryption. In this process, a partial shuffle history is exposed to the cloud. Fortunately, this leakage is minimal in reality, as is stated in the security analysis. 3) All accessed tree nodes and the desired encrypted data are packed into one package named RLL (Step E). 4) The user decrypts the RLL and obtains the final result $A[a]$. The user should reencrypt all the accessed tree nodes to reduce access-pattern leakage. 5) The user reencrypts RLL into RLL' and sends it to the cloud (Step F). 6) The cloud uses RLL' to overwrite all the accessed tree nodes (Step G). Since Steps F and G can be folded into the next request, the read operation can be implemented in a single-round-trip user-cloud interaction. The above approach can be combined with other oblivious client-computation approaches to achieving forward privacy or backward privacy.

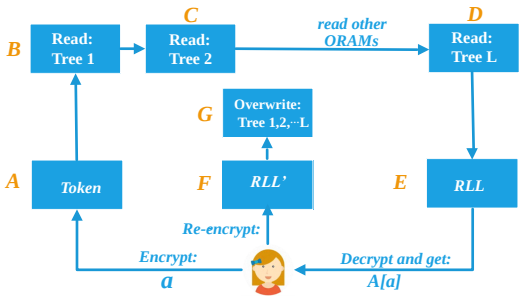

Fig. 1: Single-round-trip ORAM workflow

Randomized linked list (RLL): A randomized linked list is a sequence of accessed tree nodes. The RLL in each tree is called a randomized accessed path, which is a leaf-to-root path. To fetch a block that belongs to the leaf-to-root path, the user downloads all the nodes in the path. Each data block is related to a randomly allocated leaf identifier that is encrypted and stored in the tree node. Every time the data block is disclosed, the user generates a new random leaf identifier for the data block. Therefore, the RLLs are in random distribution.

Any user's read/write access can be viewed as a simple operation: downloading a fixed-length RLL and uploading a fixed-length RLL. The RLL is a very compact design whose encryption consists of only pseudorandom functions and symmetric encryption.

1.4 Key Contributions

1) New security definitions for forward-private ORAMs and backward-private ORAMs are given from a new perspective. These definitions are mainly designed for quantifying the query leakage.
2) We propose LL-ORAM, the first FP(BP)-ORAM that achieves single-round-trip read/write efficiency, near-zero local storage, worst-case sublinear search time, and an extremely simple implementation. LL-ORAM can help DSE solutions reduce search pattern leakage. The core procedure Algorithm 10 in Section IV consists of only several lines of pseudocode. The encryption primitives involve only Blake2b and AES.
3) LL-ORAM consists of a set of protocols that can be switched at any time for different security levels.

2 Related Work

All ORAMs are classified into large-client schemes [6], [7], [8], [9], [10], and small-client schemes [4], [5], [11], [12], [13], [14], according to storage consumed at the client side. Large-client constructions require local storage space of $O(N^c), 0 < c < 1$, and small-client constructions usually consume local space of $O(\log^2 N)$ or even less. A large-client scheme can be converted into a small-client scheme by recursively invoking itself at the cost of high client-server interactions [4]. An approach to reducing both client-side storage and client-server interactions is to adopt server-computation encryption primitives, such as garbled circuits [12]. However, the garbled circuits bring heavy computational overhead, which limits their potential in practice.

There are still some studies on write-only ORAMs [15], which say that the write is data-oblivious, and read-only ORAMs [16], which ensure only read security. Their ORAM definitions do not quantify all the possible leakages in either the read or the write. Flat ORAM [15] is a large-client application that is unscalable for large datasets. There are some studies on differentially private RAM [17], which is weaker than the traditional ORAMs, yet it is a new choice for secure data-outsourcing.

Most ORAMs have a single server, yet there are many multiserver ORAMs [18], [19], whose limitation is that they require all servers to have no collusions with each other. Sherman et al. proposed a multiclient ORAM in [20]. ORAM can be executed in parallel for more efficiency [21], and can be combined with secure hardware to improve efficiency [22], [23]. ORAMs are used in secure multiparty computation (SMC), such as [24], [25], and searchable symmetric encryption (SSE)/dynamic searchable encryption [26] (DSE).

SSE provides more features than ORAM. SSE constructions [26], [27], [28] support efficient Boolean searches, and [29] support fuzzy searches. Many SSE schemes [30], [31] provide fast range queries. Lai and Chow proposed a forward-private scheme for labeled-bipartite-graph queries in [32], which has a multidimensional leakage function for high-dimensional data. SSE encrypts outsourced data in a way that still allows the search and access pattern leakage. However, many works [33] show that the leakage can be exploited by hackers to disclose private data. Thus, we should build an efficient ORAM to reduce SSE/DSE leakage. Many recent DSE solutions focus their work on forward and backward privacy, such as [1], [27], [34], [35], [36], [37], [38], [39], [40]. Most of the works have a large-client-side storage that acts as a local index. A good approach to eliminating the storage is to employ a small-client forward-/backward-private ORAM.

3 Data Structures and Definitions

In this section, we introduce perfect ORAM security, forward ORAM privacy, backward ORAM privacy, and related definitions. Cryptographic primitives and LL-ORAM-related data structures are also given.

3.1 Privacy definitions

Full Frequency Pattern. Let Q be the list of all queries issued so far and (j, a) be the j-th entry of Q to access the logical address a. The full frequency pattern of a is defined as

$$fp(a) = \{j|(j, a) \in Q\}.$$

If Q belongs to a DSE scheme, $fp(.)$ is called the search pattern. If Q belongs to an ORAM, $fp(.)$ is called the access pattern.

Shuffle History. In an ORAM, to avoid the access frequency pattern in the logical block address a, one data block that stores the value $A[a]$ is usually moved from one physical address to another by shuffling algorithms. Assuming the ORAM has been accessed l times, the shuffle history $ShuHis(a)$ to a is defined as

$$ShuHis(a) = \{p_1, p_2, \cdots, p_l\},$$

where each p_j is the physical address for block a at the j-th ORAM-access time. Thus, the ORAM is designed to protect not only $fp(a)$ but also $ShuHis(a)$.

Stop History. Let Q' be the list of all queries issued so far, and whose j-th entry is (j, k, p_j, a), where k denotes the k-th time to access block a, and p_j is the physical address of block a. Assuming the ORAM has been accessed l times, the stop history to a is defined as

$$\begin{aligned} StopHis(a) = \{(j, p_j)|\forall j, r : (j, k, p_j, a) \in Q' \\ \wedge\ (r, k-1, p_r, a) \in Q' \\ \wedge p_j = p_r\}, \end{aligned}$$

where $p_j = p_r$ means that the data block has not been moved before the j-th ORAM access. Obviously, if the stop history always be full, this implies that the data blocks are not well shuffled with a large probability. Let Q' be the list defined above, the full frequency pattern is also written as

$$fp'(a) = \{(j, p_j)|\forall j : (j, k, p_j, a) \in Q'\}.$$

Immediate-update ORAM. Assuming the data block a has been accessed k times (k is large), an ORAM is said to be immediate-update if and only if for any block a, it holds that $|StopHis(a)| << |fp'(a)|$. The immediate-update ORAM says that the block will be moved to another place with a large probability after accessing the block.

3.2 ORAM security

Assuming all the data blocks are encrypted by randomized encryption algorithms, we analyze only the accessed addresses from the ORAM.

Perfectly secure ORAM. Let $\vec{Y}_l = ((op_1, a_1, data_1), (op_2, a_2, data_2), \cdots, (op_l, a_l, data_l))$ denote an array of operations performed by the user after ORAM initialization, where each op_j denotes a read at a logical address a_j or a write at a_j with $data_j$. Let the symbol $\stackrel{c}{\equiv}$ denote computational indistinguishability. The ORAM construction is said to be $\mathcal{L}$-secure if and only if for any data sequences $\vec{Y}_l$, there exists a probabilistic polynomial-time (PPT) simulator $\mathcal{S}$ such that

$$Info(\vec{Y}_l) \stackrel{c}{\equiv} \mathcal{S}(\mathcal{L}(\vec{Y}_l)),$$

where $Info(\vec{Y}_l)$ contains all the accessed physical addresses from the ORAM, and $\mathcal{L}(.)$ is the predefined leakage function. The ORAM is said to be perfectly secure if and only if the function can be written as $\mathcal{L}(\vec{Y}_l) = \mathcal{L}'(N, l)$, where N is the ORAM size, l is the number of ORAM operations, and $\mathcal{L}'$ is a stateless function.

Intuitively, a perfectly secure ORAM guarantees that no access frequency for any logical address is revealed during a sequence of operations. However, the perfectly secure ORAM always creates some inefficiency in reality. We study other ORAM definitions that still satisfy user requirements on some occasions.

Forward-private ORAM (FP-II). Let $\vec{Y}_l$ be the operation array defined above. For any logical address $a \in [0, N-1]$ of $\vec{Y}_l$, the leakage induced by a read or a write is described as $\mathcal{L}^{Read}(a)$ or $\mathcal{L}^{Write}(a)$, respectively. Let $\mathcal{L} = \{\mathcal{L}^{Read}, \mathcal{L}^{Write}\}$. An $\mathcal{L}$-secure ORAM is said to be forward-private (FP-II) if and only if the leakage function can be written as

$$\begin{aligned} \mathcal{L}^{Read}(a) &= \mathcal{L}'(fp(a)) \\ \mathcal{L}^{Write}(a) &= \{op\}, \end{aligned} \tag{1}$$

where $\mathcal{L}'$ is a stateless function.

An ORAM with FP-II forward privacy says that the read reveals the full frequency pattern with a nonnegligible probability, and the write is still oblivious. The FP-II ORAM has two vital properties. One is that the write is data-oblivious, which can be exploited by update-sensitive applications. The other is that there are no logical addresses are revealed. However, the FP-II ORAM has the full-frequency-pattern read leakage, which perhaps comes from the usage of repetitive read tokens.

Forward-private ORAM (FP-I). Let $\vec{Y}_l$ be the operation array defined above. For any logical address $a \in [0, N-1]$ of $\vec{Y}_l$, the leakage induced by a read or a write is described as $\mathcal{L}^{Read}(a)$ or $\mathcal{L}^{Write}(a)$, respectively. Let $\mathcal{L} = \{\mathcal{L}^{Read}, \mathcal{L}^{Write}\}$. An $\mathcal{L}$-secure immediate-update ORAM is

said to be forward-private (FP-I) if and only if the leakage function can be written as

$$\begin{aligned}\mathcal{L}^{Read}(a) &= \mathcal{L}'(His(a))\\ \mathcal{L}^{Write}(a) &= \{op\},\end{aligned} \tag{2}$$

where $\mathcal{L}'$ is a stateless function, and $His(a) \subset ShuHis(a)$.

This definition guarantees that all the write operations are oblivious and the reads can be nonoblivious, where $His(a) \subset ShuHis(a)$ denotes that only partial shuffle-history addresses are revealed in the read operation. An immediate-update ORAM will contribute to reduce the frequency pattern leakage if partial shuffle-history addresses are revealed. A forward-private ORAM implies that the write operation obliviously modified the value that will be nonobliviously read in the future. If an ORAM achieves FP-I privacy, it still has FP-II privacy, but not vice versa. We also give the backward-private ORAM definitions.

Backward-private ORAM (BP-II). Let $\overrightarrow{Y}_l$ be the operation array defined above. For any logical address $a \in [0, N-1]$ of $\overrightarrow{Y}_l$, the leakage induced by a read or a write is described as $\mathcal{L}^{Read}(a)$ or $\mathcal{L}^{Write}(a)$, respectively. Let $\mathcal{L} = \{\mathcal{L}^{Read}, \mathcal{L}^{Write}\}$. An $\mathcal{L}$-secure ORAM is said to be backward-private (BP-II) if and only if the leakage function can be written as

$$\begin{aligned}\mathcal{L}^{Read}(a) &= \{op\}\\ \mathcal{L}^{Write}(a) &= \mathcal{L}'(fp(a)),\end{aligned} \tag{3}$$

where $\mathcal{L}'$ is a stateless function.

A backward-private ORAM implies that the read operation is always oblivious even if the value has been nonobliviously written in history. The BP-II ORAM says that the read is oblivious, and the write is allowed to reveal the full frequency-pattern leakage, but no logical addresses are revealed.

Backward-private ORAM (BP-I). Let $\overrightarrow{Y}_l$ be the operation array defined above. For any logical address $a \in [0, N-1]$ of $\overrightarrow{Y}_l$, the leakage induced by a read or a write is described as $\mathcal{L}^{Read}(a)$ or $\mathcal{L}^{Write}(a)$, respectively. Let $\mathcal{L} = \{\mathcal{L}^{Read}, \mathcal{L}^{Write}\}$. An $\mathcal{L}$-secure immediate-update ORAM is said to be backward-private (BP-I) if and only if the leakage function can be written as

$$\begin{aligned}\mathcal{L}^{Read}(a) &= \{op\}\\ \mathcal{L}^{Write}(a) &= \mathcal{L}'(His(a)),\end{aligned} \tag{4}$$

where $\mathcal{L}'$ is a stateless function, and $His(a) \subset ShuHis(a)$.

The BP-I ORAM says that the read is oblivious, and the write is allowed to reveal only partial shuffle-history addresses. An immediate-update ORAM with backward privacy means the write addresses for the logical block are frequently updated.

3.3 Cryptographic Primitives

One-time random oracle. Assuming there is a random oracle $H(.)$, a one-time random oracle (OTRO) is defined as a tuple $(H, (x_1, \cdots, x_n))$ denoted $H(.)^{(n)}$, where $(x_1, \cdots, x_n)$ is an array of unique access keys performed on the oracle so far. We say $H(.)^{(n)}$ is a one-time random oracle only if for any $i \in [1, n]$, $j \in [1, n]$, and $i \neq j$, then $x_i \neq x_j$. The concept OTRO guarantees that there are no same access keys before the current time.

OTRO appears sound since it is straightforward that the outputs of the OTRO are always random values. Since encryption always involves two or more parties, it is recommended that the OTRO be instantiated with two-party (or multi-parties) encryption protocols. One party (the user) generates the unique access keys with precomputed one-time values, and the other party (the cloud) receives these keys for querying the random oracle.

One-time value. Assuming the existence of OTRO $H(.)^{(n)}$, a one-time value is defined as $H(x_i)$ ($i \in [1, n]$). From the definition, we can conclude that 1) the one-time value is indistinguishable from random by any probabilistic polynomial-time (PPT) adversary, and 2) $H(x_i)$ appears only one time among $\{H(x_i)\}_{i\in[1,n]}$.

The one-time value is an ideal concept, and it exists only on the condition that there is a random oracle. However, we assume that well-defined collusion-resistant hash functions can create one-time values. This concept plays a vital role in LL-ORAM since the overall construction consists of one-time values. The concept is not isolated, and it is always related to the OTRO.

3.4 LL-ORAM

LL-ORAM involves a set of full binary trees labeled with $(T_1, T_2, \cdots, T_L)$, where each T_i is an i-height full binary tree that consists of $(2^i - 1)$ LL-tree nodes.

LL-tree: T_L is called the data LL-tree, and each T_i ($i \in [1, L-1]$) is the position LL-tree. Given an N-block data array $A[.]$, T_L stores the data chunks in each tree node with a random order, where each chunk contains two contiguous data blocks. To save the data chunk locations, the $(L-1)$ position trees $\{T_1, \cdots, T_{L-1}\}$ are used.

Figure 2 gives a design overview of the LL-ORAM. To read a value $A[a]$, instead of scanning the full array, the algorithm only retrieves the leaf-to-root nodes of T_L. Since the algorithm does not know where the leaf identifier is, it should first search the position trees. Thus, the search begins from the first tree T_1. Since T_1 contains only one tree node, the user knows how to search T_1. The input of the i-th tree x_i comes from the $(i-1)$-th tree, where x_i is the leaf identifier of the i-th tree. If the cloud is given the token, the query will be processed by the cloud from T_1 to T_L sequentially. A read/write operation induces a set of accessed leaf identifiers $(x_1, x_2, \cdots, x_L)$. We can see that LL-trees' advantages are 1) access completes in a single-round-trip interaction, and 2) the number of accessed tree nodes is sublinear in N.

3.5 LL-tree Node

Node/leaf identifiers: Assuming there is a full binary tree, for any tree node with identifier x, its left-child identifier is encoded as $(2x+1)$, and its right-child identifier is encoded as $(2x+2)$. The advantage of the node-encoding method is that all the tree nodes share the same encoding rule.

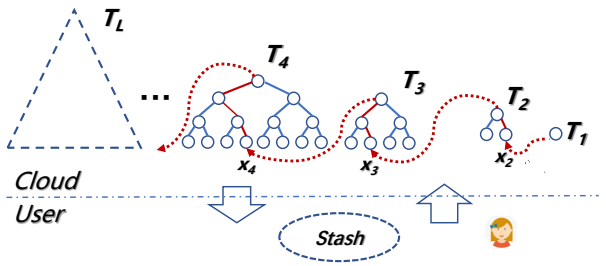

Fig. 2: A logical view of an LL-ORAM

Node structure: An LL-tree node is a composite structure consisting of two complex parts: $head$ and $tail$. Each part is defined as follows:

$$\begin{aligned} LL-treenode &= (head, tail) \\ head &= (m, emaps) \\ tail &= (M, chunks) \\ emaps &= (emap_1, emap_2, \cdots, emap_z) \\ chunks &= (chunk_1, chunk_2, \cdots, chunk_z) \\ emap_k &= (\mathsf{k}_1, \mathsf{v}_1, \mathsf{k}_2, \mathsf{v}_2), (k \in [1, z]) \\ chunk_k &= (a, data_0, data_1, leaf), (k \in [1, z]). \end{aligned}$$

The header $head$ contains $(m, emaps)$, and the tail $tail$ contains $(M, chunks)$, where m and M are encrypted values that are detailed in the next section, and $emaps$ and $chunks$ are vectors containing z items. Each encrypted map (EMap) in $emaps$ has two key-value pairs $(\mathsf{k}_1, \mathsf{v}_1, \mathsf{k}_2, \mathsf{v}_2), (k \in [1, z])$, and each chunk in $chunks$ has four parts $(a, data_0, data_1, leaf)$, where a is the address of data $data_0$, $(a+1)$ is the address of data $data_1$, and $leaf$ is the leaf identifier whose $leaf$-to-root path contains this chunk. $head$ is used by the cloud for computations but $tail$ is used only by the user to rebuild the randomized linked list (RLL). $tail$ is always in encrypted form at the cloud side, and $head.emaps$ is decrypted by the cloud only when the token matches the block.

Encrypted map (EMap): An encrypted map is a hash table that supports two types of efficient operations, writing a key-value pair (K, V) and reading a key-value pair (K, V) by the K, where K and V are fixed-size values. We use $EMap.write(K, V)$ and $V \leftarrow EMap.read(K)$ to denote a write and a read, respectively, as shown in Equations 5 and 6. The operation $EMap.write(K, V)$ has two steps: 1) K is split into two parts (K_l, K_h), and 2) $EMap[K_l] = K_h \oplus V$, assuming K_h and V share the same bit size. The operation $V \leftarrow EMap.read(K)$ also has two steps: 1) K is split into two parts (K_l, K_h), and 2) $V = EMap[K_l] \oplus K_h$. The advantage of EMap is that it takes only one pseudorandom computation to encrypt or decrypt one record.

$$EMap.write(K, V) : \begin{cases} K \rightarrow (K_l, K_h) \\ EMap[K_l] \leftarrow K_h \oplus V \end{cases} \quad (5)$$

$$V \leftarrow EMap.read(K) : \begin{cases} K \rightarrow (K_l, K_h) \\ V \leftarrow K_h \oplus EMap[K_l] \end{cases} \quad (6)$$

If the accessing key K is a one-time value, the value V is encrypted by K. Since K can be used only once after V is disclosed, we rebuild the whole EMap. To easily reencrypt the EMap in every data access, we put only one chunk that contains only two data blocks in one EMap.

Data chunk: The data chunk, which belongs to the tail part of the node, has two key-value pairs and a leaf identifier. Each data chunk saves two contiguous data blocks. We call $data_0$ the first data block with address a and $data_1$ the second data block with address $(a+1)$. Each chunk is related to a leaf, whose identifier is $leaf$. The $leaf$-to-root path always contains this data chunk since the user initializes the value. The value $leaf$ is only reserved for node shuffling at the user side. Before the LL-tree node is outsourced, the whole tree node is encrypted by RCPA-secure algorithms, such as the counting-mode AES. In the next section, we show how to put the data chunks into the EMaps.

There are three tree node states: 1) empty, 2) padded with dummy values and encrypted, and 3) full and encrypted. Initially, all the tree nodes are in State 1. After several data accesses, many accessed nodes turn into State 2. Since an L-height tree is allowed to hold $2^L - 1$ real data blocks at most, there are still many dummy blocks. For security concerns, States 2 and 3 cannot turn into State 1.

3.6 Address Encoding

Intuitively, a small ORAM, in general, is more efficient than a large ORAM. For efficiently accessing the ORAM, an address is converted into a set of smaller values.

Given an address a, the algorithm $Convert$ is defined as follows:

$$\begin{aligned} a_i &= a_{i+1}/2, \\ b_i &= a_{i+1}\%2 \ for \ all \ i \in [1, L], \end{aligned} \quad (7)$$

where a_i is an integer, b_i is a bit, and $a_{L+1} = a$. Through Equation 7, address a is converted into a set of L pairs. We write the algorithm as $Convert(a) \rightarrow \{(a_1, b_1), (a_2, b_2), \cdots, (a_L, b_L)\}$, where each (a_i, b_i) is called a block address, an integer of $(2a_i + b_i)$.

3.7 Stash

A stash is a storage structure to hold or evict retrieved data chunks. There are two types of stashes. One is the temporary stash that is stored at the user side, and the other is the root stash that is the root node of the trees. LL-ORAM uses only a temporary stash to shuffle returned blocks, and all the blocks are reencrypted and sent back to the cloud immediately. The temporary stash is a hash table using the following data structure,

$$\begin{aligned} Stash &= (sblock_1, sblock_2, \cdots) \\ sblock_j &= (a', (data_0, data_1, leaf)). \end{aligned}$$

Each block in $Stash$ can be accessed by the address a' and one bit b, and a block $(data_b, leaf)$ is matched. If b is zero, $data_0$ is selected, otherwise $data_1$ is used. $data_0$ and $data_1$ are arbitrary data, and $leaf$ denotes a leaf identifier whose corresponding $leaf$-to-root nodes contain the block.

4 LL-ORAM: DETAILS AND IMPLEMENTATIONS

This section presents the tree-encryption algorithm, the data-access algorithm, the shuffle algorithm, and four communication protocols, including the single-round-trip access oblivious-access protocol, the forward-private protocol, and the backward-private protocol.

4.1 An overview

Data distribution. In the initial stage, all the trees are empty. After any query, for any chunk $(a', data_0, data_1, leaf)$ in any position LL-tree T_i ($i > 1$), the value $leaf$ is always stored in the prior small tree T_{i-1}, whose corresponding chunk is $(a'/2, data_0^*, data_1^*, leaf^*)$. If $a'\%2 = 0$, then $data_0^* = leaf$ is selected, else $data_1^* = leaf$. Note that even if the tree has been shuffled, the above rule always holds.

Roadmap. Subsection B gives the tree-node encryption algorithm, which happens on the client side. Subsection C shows how to search for an address. Since any chunk can be located by applying the above rule recursively, the cloud can always find the correct leaf-to-root path containing the final data block. Subsection D elaborates on the client-side eviction algorithm and the client-side data read/write. Subsection E details four LL-ORAM communication protocols.

4.2 LL-tree Encryption

Node mask: An LL-tree node mask is a random value generated by the user for encrypting the LL-tree node. Each LL-tree node has a unique mask that is encrypted and stored in the node's tail part. Given an LL-tree node d, the mask is denoted by $d.tail.M$. In our experiments, the mask is a 20-byte random value. After any access to the data block, the user will entirely rebuild the whole tree node and its mask. Note that the mask is secret, and it does not exist alone in the cloud. It is not revealed to the cloud all the time.

Encrypted mask (EMask): If two masks are XORed with each other, we call the XORed mask EMask. For any LL-tree node d, assuming its father-node mask is M_f, we denote $d.head.m$ the EMask, where $d.head.m = d.tail.M \oplus M_f$. EMask is the core design of the LL-tree.

Block key: A block key is a key to encrypting the data block. Let the data block be $data$ with the block address (a_i, b_i), where i corresponds to the i-th LL-tree. Let F be a keyed hash function modeled as a random oracle, $\mathbb{K}$ be the user's secret key, and M be the current node mask. The block key is computed by $F_{\mathbb{K}}(i||a_i||b_i) \oplus M$. The $BlockKey$ algorithm is shown in Algorithm 1. From this, we can conclude that for any LL-tree, any tree node, and any data block, the block key is unique. Since each chunk has two blocks, then a chunk has two block keys.

Algorithm 1 Block key generation.

BlockKey(i, (a_i, b_i), M):

1) return $bk \leftarrow F_{\mathbb{K}}(i||a_i||b_i) \oplus M$

Node encryption: The LL-tree node encryption pseudocode is shown in Algorithm 2. Let i denote the i-th tree. Given an unencrypted tree node d, whose mask is M, and whose father-node mask is M_f, the algorithm outputs an encrypted tree node d, assuming all the unencrypted chunks have been put into $d.tail.chunks$ in advance.

First, the algorithm encrypts $d.head$. Since $d.head.m \leftarrow M \oplus M_f$, M and M_f are encrypted. The structure $d.head.emaps$ stores Z EMaps, and each Emap saves two contiguous data blocks. The k-th EMap is denoted as $d.head.emaps[k]$. To put a data block into $d.head.emaps[k]$, the user should generate the corresponding block key first. For each unencrypted chunk $chunk$ in $d.tail.chunks$, the user builds the block keys $bk_0 \leftarrow BlockKey(i, (chunk.a, 0), M)$ and $bk_1 \leftarrow BlockKey(i, (chunk.a, 1), M)$. With the block keys, the blocks can be written into the EMaps with $d.head.emaps[k].write(bk_0, chunk.data_0)$ and $d.head.emaps[k].write(bk_1, chunk.data_1)$. Through the above method, $2Z$ data blocks are put into $d.head.emaps$.

Second, the algorithm encrypts $d.tail$. The user puts M into $d.tail.M$, which is reserved for node rebuilding. The user utilizes an RCPA-secure private-key algorithm to encrypt the whole part $d.tail$ that is always hidden from the cloud.

Tree-root encryption differs from that of its children. We assume that the root still has a father, and its mask is held by the user as the secret value, called the user mask. If the user always stores a global variable $Counter$ to save the total number of reads/writes performed on the LL-ORAM, the user masks are generated with the following approach. Let M_i be the i-th user mask, i.e., the father mask of the root of the i-th tree, $\mathbb{K}$ be the user's secret key, and G be a keyed collusion-resistant pseudorandom function. The set of user secret masks is $\{M_1, M_2, \cdots, M_L\}$, where each $M_i = G_{\mathbb{K}}(i||Counter)$.

Algorithm 2 LL-tree node encryption.

NodeEncrypt(d, i, M, M_f):

1) $d.tail.M \leftarrow M$
2) $d.head.m \leftarrow M \oplus M_f$
3) for $k = 0$ to $d.tail.chunks.size() - 1$
   a) $chunk \leftarrow d.tail.chunks[k]$
   b) $bk_0 \leftarrow BlockKey(i, (chunk.a, 0), M)$
   c) $d.head.emaps[k].write(bk_0, chunk.data_0)$
   d) $bk_1 \leftarrow BlockKey(i, (chunk.a, 1), M)$
   e) $d.head.emaps[k].write(bk_1, chunk.data_1)$
4) encrypt $d.tail$
5) return d

Encrypted linked list (EL): All the tree nodes of the x_i-to-root path form a list named the encrypted list, denoted EL_x^i, which is a set of (leaf x)-to-root tree nodes in T_i. Note that the encrypted list is just a logical concept.

Figure 3 shows an example of the encrypted linked list, which is part of the LL-tree (T_4). There are four nodes $\{d_A, d_B, d_C, d_D\}$ in the linked list, with the corresponding node masks $\{M_A, M_B, M_C, M_D\}$. Note that the masks are personal values, and the cloud does not learn them forever. Assume d_C contains an encrypted block with the block address (a_i, b_i), and the block is the first position of d_C. The cloud wants to obtain a value that corresponds to the address (a_i, b_i).

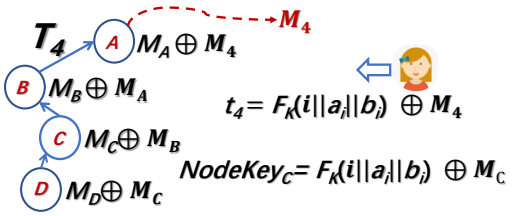

Fig. 3: Encrypted linked list in an LL-tree (L=4).

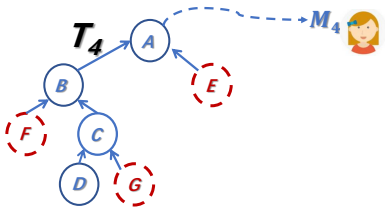

Fig. 4: Sibling encrypted linked list in an LL-tree (L=4).

The user sends a token $t_4 \leftarrow F_{\mathbb{K}}(4||a_i||b_i) \oplus M_4$ to the cloud. The cloud computes $t_4 \oplus (d_A.head.m) = F_{\mathbb{K}}(4||a_i||b_i) \oplus M_4 \oplus M_A \oplus M_4 = F_{\mathbb{K}}(4||a_i||b_i) \oplus M_A$, and this is, in fact, one of the block keys of d_A for searching (a_i, b_i). Since d_A does not contain the address, the cloud continues to search d_B. It computes $t_4 \oplus (d_A.head.m) \oplus (d_B.head.m)$, and it obtains nothing from d_B. Then, it searches d_C, and computes $t_4 \oplus (d_A.head.m) \oplus (d_B.head.m) \oplus (d_C.head.m) = F_{\mathbb{K}}(4||a_i||b_i) \oplus M_C$. It continues to access d_C's EMap, and computes $V \leftarrow d_C.head.emaps[0].read(F_{\mathbb{K}}(4||a_i||b_i) \oplus M_C)$. Fortunately, the cloud obtains the final value V. As it is hoped, V equals $d_C.tail.chunks[0].data_{b_i}$.

The cloud can access any encrypted list EL_x^i with the tree number i and the leaf identifier x. However, without the user's tokens for any chunks, EL_x^i cannot be decrypted by the cloud. A user's token reveals exactly only one block address in each tree.

Cloud computing: Let t_i be the user's token of the i-th tree, i.e., $t_i = F_{\mathbb{K}}(i||a_i||b_i) \oplus M_i$. An encrypted linked list EL_x^i can be efficiently computed with the following algorithm. Algorithm 3 takes as input EL_x^i and t_i, and it outputs $y \leftarrow Compute(EL_x^i, t_i)$. This algorithm contains only two loops: 1) accessing each tree node of EL_x^i, and 2) generating the block key bk that is used for accessing the encrypted map in each node.

Algorithm 3 Computation in the cloud

Compute(EL_x^i, t_i):

1) initialize $bk \leftarrow t_i$
2) for all d in EL_x^i
   a) $bk \leftarrow bk \oplus (d.head.m)$
   b) for all $EMap$ in $d.head.maps$
      i) $y \leftarrow EMap.read(bk)$
      ii) if y is empty, continue;
      iii) else, return y.

Sibling encrypted linked list (SL): Given a tree node d, the sibling node of d is written as $sn(d)$. In the special case, the sibling node of the root is empty. A sibling encrypted linked list is defined as $SL_x^i = \{sn(d) | d \in EL_x^i\}$.

Consider an internal tree node d. When d has been rebuilt, two children nodes whose EMasks are related to d should also be rebuilt. Thus, ELs and SLs should be downloaded and reencrypted by the user. Figure 4 shows an SL that contains only three nodes $\{E, F, G\}$. If $\{A, B, C, D\}$ are accessed and rebuilt, nodes $\{E, F, G\}$ should also be updated.

Randomized linked list (RLL): To access an address a on an LL-ORAM, all accessed tree nodes along with their siblings in the query form a long linked list called a randomized linked list. The user's token is $(t_1, t_2, \cdots, t_{L-1})$, where each $t_i = F_{\mathbb{K}}(i||a_i||b_i) \oplus M_i$. We define RLL=$\{(EL_{x_1}^1, SL_{x_1}^1), \cdots, (EL_{x_{L-1}}^{L-1}, SL_{x_{L-1}}^{L-1}), (EL_{x_L}^L, \{\})\}$, where each $x_{i+1} = Compute(EL_{x_i}^i, t_i), (i \in [1, L-1])$. Note that the token of the last tree t_L cannot be sent to the cloud; otherwise, the cloud learns the final result, which is the private user's data $A[a]$. Since $EL_{x_L}^L$ has not been disclosed, $SL_{x_L}^L$ does not need to be rebuilt.

An RLL contains len=$\sum_{i=1}^{L-1}(i + i - 1) + L$=$L^2 - L + 1$ LL-tree nodes. All the RLLs are a fixed length.

4.3 Searching for an Address

LL-trees can be efficiently searched by the user's token that encodes an address into a set of one-time values. The cloud sends back all the accessed tree nodes to the user for node shuffling and reencrypting.

One-time token: A one-time token is a set of one-time values that can be sent by the user only once. Figure 4 shows how to create a one-time token from the address a. If M_i is a randomized one-time number of T_i, the user's token of the i-th tree can be written as $t_i = F_{\mathbb{K}}(i||a_i||b_i) \oplus M_i$. The full token tk for $(T_1, T_2, \cdots, T_{L-1})$ equals $(t_1, t_2, \cdots, t_{L-1})$.

Algorithm 4 Building a one-time token

BuildToken(a):

1) $Convert(a) \rightarrow ((a_1, b_1), (a_2, b_2), \cdots, (a_L, b_L))$
2) for $i = 1$ to $L - 1$
   a) $t_i \leftarrow F_{\mathbb{K}}(i||a_i||b_i) \oplus M_i$.
3) $tk \leftarrow (t_1, t_2, \cdots, t_{L-1})$
4) return tk

Searching for an address: When the cloud owns the one-time token, the cloud can individually access each tree with the algorithm $Search$ shown in Algorithm 5. The cloud sequentially touches $\{T_1, \cdots, T_L\}$ using the function $Compute$. All the accessed ELs and SLs are packed into rll and reserved for rebuilding the corresponding tree nodes later.

The search algorithm outputs an RLL and a set of randomly-accessed addresses $\{x_1, x_2, \cdots, x_L\}$. The randomness comes from the following RLL-rebuilding algorithm.

4.4 RLL Rebuilding

We present the client-side shuffle algorithm for RLL rebuilding. Rebuilding all accessed tree nodes is necessary.

Algorithm 5 Searching for an address

Search(tk):

1) initialize $rll \leftarrow \{\}$
2) $x \leftarrow 0$
3) for all $i = 1$ to $L - 1$
   a) $x \leftarrow Compute(EL_x^i, tk.t_i)$
   b) $rll \leftarrow rll \cup \{(EL_x^i, SL_x^i)\}$
4) $rll \leftarrow rll \cup \{(EL_x^L, \{\})\}$
5) return rll

Otherwise, the full frequency pattern leakage is induced. Algorithms 6, 7, and 8 show how to rebuild an EL, an SL, and an RLL, respectively.

EL rebuilding: Rebuilding an EL takes as input the old EL EL_x^i, op 'write' or 'read', (a^*, b^*) a block address, x^* a newly allocated randomized leaf identifier, and y the arbitrary data to write. $RebuildEL$ has three purposes: 1) to rebuild the EL to avoid the full frequency pattern leakage, 2) to write data y with the block address (a^*, b^*) into the ORAM, and 3) to read data with address (a^*, b^*).

The algorithm consists of eight steps, as shown in Algorithm 6. In Step 1, EL_x^i is delivered back to the user with the leaf identifier x and the tree number i. The cloud will use x to shuffle the returned tree nodes and use (i, x) to overwrite the tree nodes. In Step 2, the user renews a new one-time value M_i, one of the user's values for the i-th tree T_i. In Step 3, all chunks in EL_x^i are decrypted and put into the local stash $Stash$, which uses a local hash table to hold all chunks indexed by key (a', b'), where a' is the block address, and b' is the choice of the first or the second. Note that each chunk has a leaf identifier to shuffle. In Step 4, the user writes the new data y as well as its leaf identifier x^* into $Stash$, i.e., $Stash.write((a^*, b^*), (x^*, y))$. Note that the user cannot overwrite the existing block whose address is $(a^*, NOT\ b^*)$. In Step 5, the user creates an empty EL el to hold the returned result. In Step 6, the user generates L random masks $\{M_{i,1}, \cdots, M_{i,L}\}$ for L nodes in el of the i-th tree. The user stores these masks into the tree nodes. In Step 7, the user shuffles the chunks by using an algorithm. $\mathcal{P}(x, l)$ denotes an l-level-node-to-root path that is a substring of the x-to-root path. From level L to level 1, sequentially shuffle the chunks. For each chunk s in the stash at level l, $\mathcal{P}(x, l) = \mathcal{P}(s.x, l)$ means that these chunks will be moved to the new level l. Each tree node can hold Z chunks, yet the root can hold S chunks. If the node is full, the remains will be evicted into the upper levels. In level 1, all the remaining chunks in $Stash$ will be evicted into the root node. Note that after the eviction, the user saves nothing blocks in $Stash$. If the number of encrypted chunks in the root is less than S, the root should be padded with dummy chunks to size S. In Step 8, the new EL el with the address (a^*, b^*) and the data (x^*, y) is generated. Since the el is encrypted by the RCPA-secure algorithm and the pseudorandom function, EL leaks no privacy.

An EL has a sibling SL. If the EL masks have been updated, rebuilding the SL is necessary since all the SL EMasks have been changed. Otherwise, the linked lists do not work.

SL rebuilding: Rebuilding the SL involves only the operation to update the EMasks, as shown in Algorithm 7. The father-mask values come from Step 6) of Algorithm 6. For each node d in SL, $d.head.m$ is updated to the new EMask, i.e., $d.head.m \leftarrow (d.tail.M) \oplus M_{i,j-1}$, where $M_{i,j-1}$ is the father mask of node d. The tail information $d.tail$ remains unchanged. Note that since the SL is always undisclosed in the cloud, the node masks in the EL do not need to be updated.

Algorithm 6 EL rebuilding

RebuildEL(EL_x^i, op, (a^*, b^*), x^*, y):

1) parse EL_x^i as (EL, i, x);
2) $Counter \leftarrow Counter + 1$;
3) $M_i = G_{\mathbb{K}}(i||Counter)$; assume $M_{i,0} = M_i$
4) $Stash \leftarrow$ Read all chunks from EL_x^i
5) if $op =$ '$write$', $Stash.write((a^*, b^*), (x^*, y))$;
6) initialize $el \leftarrow \{\}$
7) for $l = L$ to 1, $M_{i,l} \leftarrow Random()$;
8) for $l = L$ to 1
   a) $S' \leftarrow \{\}$
   b) for all s in $Stash$
      i) if $\mathcal{P}(x, l) = \mathcal{P}(s.x, l)$, $S' \leftarrow S' \cup s$
   c) if $l > 1$, $S' \leftarrow$ Select $min(|S'|, Z)$ chunks from S'
   d) else, use entire S';
   e) $Stash \leftarrow Stash - S'$
   f) create a new l-level node d, and put S' into d
   g) $el \leftarrow el \cup NodeEncrypt(d, i, M_{i,l}, M_{i,l-1})$
9) return el

Algorithm 7 SL rebuilding

RebuildSL(SL_x^i):

1) parse SL_x^i as (SL, i, x)
2) $sl \leftarrow \{\}$
3) for all $d \in SL$
   a) read the level of d, i.e., $j \leftarrow d.level$
   b) Assume the new mask $M_{i,j-1}$ has been globally stored by the user
   c) decrypt $d.tail$
   d) $d.head.m \leftarrow (d.tail.M) \oplus M_{i,j-1}$
   e) encrypt $d.tail$
   f) $sl \leftarrow sl \cup d$
4) return sl

RLL rebuilding: Rebuilding an RLL requires the following parameters: the old RLL, op 'write' or 'read', an address a, and $data^*$, the arbitrary data to write. RLL rebuilding has three purposes:

- To rebuild the RLL to reduce access pattern leakage;
- To write $data^*$ with address a into the ORAM;
- To read data with address a.

Rebuilding the RLL contains the operations to reencrypt all the ELs and SLs of the original RLL. As shown in Algorithm 8, the algorithm includes seven steps. In Step 1, address a is converted into a set of block addresses. In Step 2, the user searches from $EL_{x_L}^L$ that contains the final data $data$ with the block address (a_L, b_L). If

op = '$read$', $data$ is returned. In Step 3, L new random leaf addresses $\{x_1^*, x_2^*, \cdots, x_L^*\}$ are allocated by invoking $RandomLeaf(i)$, which outputs the random leaf identifier for the i-th tree with the node-encoding method. These addresses are the new leaf identifiers for the next oblivious read/write query for address a, yet currently, the identifiers are still hidden from the cloud. In Step 4, the user builds an empty RLL^* to hold the results. In Step 5, the user invokes $RebuildEL$ to rebuild the EL in the data tree T_L, puts $(x_L^*, data^*)$ into T_L, invokes $RebuildEL$ to rebuild the EL in the position trees $\{T_{L-1}, T_{L-2}, \cdots, T_1\}$, and stores the new leaves $\{x_{L-1}^*, x_{L-2}^*, \cdots, x_1^*\}$. In Step 6, the user rebuilds all SLs $\{SL_x^1, \cdots, SL_x^L\}$. Since the user renewed all the masks in the ELs, the user can reencrypt SLs locally. In Step 7, the user obtains the result $data$ and rll', which is the new RLL.

Algorithm 8 Reencrypting an RLL

RebuildRLL(RLL, op, a, $data^*$):

1) $Convert(a) \rightarrow ((a_1, b_1), \cdots, (a_L, b_L))$
2) $data \leftarrow$ Read data by (a_L, b_L) from $RLL.EL_{x_L}^L$
3) for $i = 1$ to L, $x_i^* \leftarrow RandomLeaf(i)$;
4) initialize $RLL^* \leftarrow \{\}$
5) for $i = L$ to 1
   a) $el \leftarrow RLL.EL_{x_i}^i$
   b) if $i = L$,
   c) $el' \leftarrow RebuildEL(el, op, (a_L, b_L), x_i^*, data^*)$
   d) else,
   e) $el' \leftarrow RebuildEL(el, 'write', (a_i, b_i), x_i^*, x_{i+1}^*)$.
   f) $RLL^*.EL_{x_i}^i \leftarrow el'$
6) for $i = L$ to 1
   a) $RLL^*.SL_{x_i}^i \leftarrow RebuildSL(RLL.SL_{x_i}^i)$
7) return $(RLL^*, data)$

Overwriting nodes: After node regeneration, the cloud puts all the new tree nodes into the original paths. As shown in Algorithm 9, for any LL-tree T_i, all the tree nodes from the original x_i-to-root path are overwritten, where T_i denotes the i-th tree, $T_i[x, l]$ denotes an l-level node at the x-to-root path in the i-th tree, and $d.level$ denotes the level of node d. Note that in the overwriting procedure, since the RLL is fixed-length $(L^2 - L + 1)$, no leakage is induced.

Algorithm 9 Overwriting an RLL

Overwrite(RLL):

1) for $i = 1$ to L
   a) parse $RLL.EL_x^i$ as (EL, i, x)
   b) for all d in EL
      i) overwrite $T_i[x, d.level] \leftarrow d$
   c) parse $RLL.SL_{x'}^i$ as (SL, i, x')
   d) for all d' in SL
      i) overwrite $T_i[x', d'.level] \leftarrow d'$

4.5 LL-ORAM protocols

We set up four protocols: a single-round-trip protocol, an oblivious protocol, a forward-private protocol, and a backward-private protocol. The user has four options to interact with the cloud at any time.

LL-SR-ORAM: LL-SR-ORAM is a single-round-trip access protocol denoted LL-SR-ORAM=$\{Access\}$. Algorithm 10 provides the single-round-trip API to access one data block. The protocol sequentially invokes the subroutines $BuildToken$, $Overwrite$, $Search$, and $RebuildRLL$. If op ='$read$', the input is ('$read$', a, 0), and returns $A[a]$. If op ='$write$', the input is ('$write$', a, $data^*$). The small trick in Algorithm 10 is that the user creates a cache to hold rll', which is the last RLL to overwrite the original tree nodes. In the next query, the user sends the token and rll' to the cloud. Before any tree access, the cloud overwrites the existing tree nodes with rll' first.

Algorithm 10 Single-round-trip access protocol (LL-SR-ORAM)

Access(op, a, $data^*$):

1) ***User:***
   a) $tk \leftarrow BuildToken(a)$
   b) send (tk, rll') to the cloud
2) ***Cloud:***
   a) $Overwrite(rll')$
   b) $rll \leftarrow Search(tk)$
3) ***User:***
   a) $(rll', data) \leftarrow RebuildRLL(rll, op, a, data^*)$
   b) save rll' locally.
   c) if $op =$ '$read$', return $data$

Algorithm 11 Oblivious access protocol (LL-OB-ORAM)

ObAccess(op, a, $data^*$)

1) **User:**
   a) $\{t_1, \cdots, t_{L-1}\} \leftarrow BuildToken(a)$
   b) initialize $rll \leftarrow \{\}$
   c) $x \leftarrow 0$
   d) send the cached rll' to the cloud
2) **Cloud:** $Overwrite(rll')$
3) for $i = 1$ to $L - 1$
   a) **User:** send x and i to the cloud
   b) **Cloud:** send EL_x^i and SL_x^i to the user
   c) **User:** $x \leftarrow Compute(EL_x^i, t_i)$
   d) **User:** $rll \leftarrow rll \cup \{(EL_x^i, SL_x^i)\}$
4) **User:**
   a) $rll \leftarrow rll \cup \{(EL_x^L, \{\})\}$
   b) $(rll', data) \leftarrow RebuildRLL(rll, op, a, data^*)$
   c) save rll' locally.
   d) if $op =$ '$read$', return $data$

LL-OB-ORAM: LL-OB-ORAM is an oblivious-access protocol denoted LL-OB-ORAM=$\{ObAccess\}$, as shown in Algorithm 11. In the protocol, instead of sending the token to the cloud, the user sends only the leaf identifiers $x = x_i$ $(i \in [1, L])$ in every step. Since each EL_x^i is computed by the user, $Compute$ incurs no leakage. Except for $O(\log N)$ interactions, network latency, and security, the efficiency of LL-OB-ORAM is almost the same as LL-SR-ORAM. LL-

OB-ORAM achieves obliviousness at the cost of $O(\log N)$ interactions.

Algorithm 12 Forward-private ORAM protocol (LL-FP-ORAM)

FpRead(a):

1) return Access('read', a, $\perp$)

FpWrite(a, $data^*$):

1) ObAccess('write', a, $data^*$)

Algorithm 13 Backward-private ORAM protocol (LL-BP-ORAM)

BpRead(a):

1) return ObAccess('read', a, $\perp$)

BpWrite(a, $data^*$):

1) Access('write', a, $data^*$)

LL-FP-ORAM/LL-BP-ORAM: LL-FP-ORAM is a forward-private ORAM protocol denoted LL-FP-ORAM= $\{FpRead, FpWrite\}$, whose pseudocode is shown in Algorithm 12. LL-BP-ORAM is a backward-private ORAM protocol denoted LL-BP-ORAM=$\{BpRead, BpWrite\}$, whose pseudocode is shown in Algorithm 13. A forward-private ORAM or a backward-private ORAM appears if the single-round-trip protocol and the oblivious access protocol are combined.

5 Improvements and Applications

In this section, we adjust the block size to reduce the communication bandwidth and improve the search efficiency. We also study the applications to dynamic searchable encryption.

5.1 Communication Bandwidth

Equation 8 shows the communication bandwidth $band$, which approximately equals the bit size of an RLL. In a read/write operation, the communication bandwidth consists of two parts. One is the RLL, and the other is the one-time token. The token size is only $O(L)$ bytes, which can be ignored compared with the RLLs. Let us consider only the RLL.

Assuming $N = (2^L - 1)$ real blocks exist in the ORAM, except for the root, each node has Z chunks including the dummy nodes, each chunk has two blocks, each data block has B bits, each position block has B' bits, and each root has $O(BS)$ bits, the size of the RLL is computed by:

$$\begin{aligned} band_d &= (L-1)O(BZ) + O(BS) \\ band_p &= \sum_{i=1}^{L-1}((i-1)O(B'Z) + O(B'S)) \\ band &= band_d + band_p, \end{aligned} \tag{8}$$

where $band_d$ is the bit size of $(EL^L_{x_L}, \{\})$, and $band_p$ is the bit size of $\{(EL^1_{x_1}, SL^1_{x_1}), \cdots, (EL^{L-1}_{x_{L-1}}, SL^{L-1}_{x_{L-1}})\}$.

5.2 Stash Size

In our experiments, the root-stash size exceeding S can be negligible, only if Z and S are large enough. In general, Z is set to 3, S is set to 7, and $B' \approx \log N$. Equation 8 can be written as $R = O(B \log N + \log^3 N)$ bits. If $B = \Omega(\log^2 N)$ is adopted, the bandwidth overhead equals $O(B \log N)$, which is the asymptotically optimal bandwidth.

5.3 Block Size and Cloud Storage

We adjust the data block size B to obtain more data in each access. There are two types of blocks: data blocks in T_L and position blocks in $T_i (i \in [1, L-1])$. Assuming there are N data blocks, each tree node (except for the root) contains Z chunks, each root has S chunks, each chunk has two blocks, each position block has B' bits, and each data block has B bits that are a set of r contiguous records, the storage space of the LL-ORAM is computed by

$$\begin{aligned} size_d &= (N-1)O(BZ) + O(BS), \\ size_p &= \sum_{i=1}^{\log N - 1}((2^i - 1)O(B'Z) + O(B'S)), \\ size &= size_d + size_p \end{aligned} \tag{9}$$

where $size_d$ is the storage size of the data LL-tree, and $size_p$ is the size of the position LL-trees. Therefore, the data array $A[.]$ stores $N \cdot r$ data records in total. Note that the data record is an arbitrary fixed-size data structure.

In reality, we set $Z = 2$, $B' = \log N$, and $S = 7$. We have $size = O(BN + \log^3 N)$ bits. If $B = \Omega(\log^2 N)$, the LL-ORAM cloud storage is of $size = O(BN)$ bits.

5.4 Client Storage and Access Time

The user stores the permanent secret key $\mathbb{K}$ and a temporary RLL. If $B = \Omega(\log^2 N)$ is adopted, the communication bandwidth for each query is $O(B \log N)$ bits. When the user is busy, the client storage is $O(B \log N)$ bits since the user always holds the last RLL for the next data access. Note that when the cloud and the user are free, the user can upload the last RLL to the cloud. Therefore, client storage is $O(1)$. The access time is $O(\log^2 N)$, which is determined by the number of noncontiguous memory accesses.

5.5 Applications to Dynamic Searchable Encryption

One can easily implement an inverted index for dynamic searchable encryption by making black-box use of an FP(BP)-ORAM. The approaches are introduced in many works, such as [23]. Since the partial shuffle-history pattern is smaller than the full frequency pattern, LL-ORAM can improve the DSE security.

6 Security Analysis

6.1 Security Components

Masks. The user's masks are one-time values. If G is modeled as a random oracle, the i-th mask $M_i = G_{\mathbb{K}}(i||Counter)$ is indistinguishable from random after every data access. In our settings, M_i is a 20-byte integer that is enough to protect the user's privacy. The node mask, which is related to the tree node, is also a random number generated by the user every time the node is accessed.

From the above, we conclude that all the masks are random numbers.

Tokens. The user's tokens are one-time values. If F and G are modeled as random oracles, the token $t_i = F_{\mathbb{K}}(i||a_i||b_i) \oplus M_i$ is a one-time value. In the token, M_i does not exist alone. It is XORed by the value $F_{\mathbb{K}}(i||a_i||b_i)$, which is the one-time key to obtain the data block (a_i, b_i) that is stored in the cloud. Without t_i, the cloud cannot locate where the data block (a_i, b_i) is. After the cloud accesses this data block, the user renews M_i with another random number. Next, we show that all the masks in all the trees are secure enough.

ELs and SLs. An encrypted linked list (EL) is a set of leaf-to-root LL-tree nodes, denoted EL^i_x. In the i-th tree, since a user query discloses only one data block, other tree nodes remain untouched. The node mask $M_{i,j}$, which belongs to the i-th tree and the j-level node d, is always XORed by the father-node mask $M_{i,j-1}$ with the encrypted form $d.head.m$. The cloud learns only the (node x)-to-root EMasks $(m_L, m_{L-1}, \cdots, m_1)$ and t_i,

$$\begin{aligned} m_L &= M_{i,L} \oplus M_{i,L-1} \\ &\cdots \\ m_1 &= M_{i,1} \oplus M_i \\ t_i &= F_{\mathbb{K}}(i||a_i||b_i) \oplus M_i \end{aligned} \tag{10}$$

In Equation 10, there are $(L+2)$ variables in the right part, yet the cloud learns only $(L+1)$ numbers in the left part. The only way to learn one of the variables is to learn one of the other variables first, which is a paradox. To hide the access pattern of this leaf identifier x, after accessing EL^i_x, the user rebuilds EL^i_x and resets the masks to other random values. In the next query, the cloud still faces the same paradox of 'seeking $(n+1)$ variables from n numbers'. From the above analysis, we conclude that the ELs leak nothing of access patterns.

To enable the EL to work well in the next query, all pointers that point to the EL should be rebuilt. This is the work of rebuilding SL. An SL SL^i_x is the set of sibling tree nodes of the nodes in EL^L_x. In the rebuilding process, the tail does not need to be rebuilt. For a node d in SL^i_x, the user rebuilds only $d.head.m$ in the next query. $d.tail.M$ is always encrypted by an RCPA-secure algorithm, which guarantees that the outputted value cannot be distinguished from random by any PPT adversary. To learn the node mask from $d.head.m$, the cloud still faces the paradox of Equation 10. From the above, we conclude that all node masks are hidden from the cloud.

EMaps. EMap security is guaranteed by the randomness of the key. An EMap is a hash table that maps a key K to a value V for storing data $EMap.write(K, V)$, or for retrieving data $V \rightarrow EMap.read(K)$, as shown in Equations 5 and 6. The key K is used to XOR the other part value and to play the role of the accessing key. Assuming the node d contains a position block (a_i, b_i) encrypted by an EMap, K is, in fact, the block key $F_{\mathbb{K}}(i||a_i||b_i) \oplus M_d$, where M_d is the mask of d. Without K, no PPT adversary can obtain V from the $EMap$.

LL-trees and RLLs. The node encryption algorithms guarantee LL-tree security. In a tree node d, $d.tail$ is always undisclosed in the cloud, and the tail is only reserved for node rebuilding. The cloud touches only $d.head$, which consists of $d.head.maps$ and $d.head.m$.

An RLL is a set of ELs and SLs, $\{(EL^1_{x_1}, SL^1_{x_1}), \cdots, (EL^{L-1}_{x_{L-1}}, SL^{L-1}_{x_{L-1}}), (EL^L_{x_L}, \{\})\}$. Each data access generates a set of leaf identifiers $(x_1, x_2, \cdots, x_L)$ that is computed by the cloud. The strength of the RLL is that $(x_1, x_2, \cdots, x_L)$ are random values that are the user's predefined leaf positions in each query.

6.2 Leakage Analysis

We define the leakage function only for $Access(.)$ since LL-SR-ORAM, LL-FP-ORAM, and LL-BP-ORAM use this API. Let $\overrightarrow{Q_l} = (a_{i,1}, a_{i,2}, \cdots, a_{i,l})$ be a sequence of logical addresses to access the position LL-tree T_i. For simplicity, let $a = a_{i,l}$, which denotes the l-th logical address that corresponds to a block address. The leakage function of $Access(.)$ for T_i is defined as

$$\mathcal{L}(T_i, a) = \{(x_{i,a}, y_{i,a})\}, \tag{11}$$

where $(x_{i,a}, y_{i,a})$ is the last shuffle address for storing block a, $x_{i,a}$ is the leaf identifier, and $y_{i,a}$ is the height of the stored block in T_i. Given a logical address a to access T_i, $\mathcal{L}(T_i, a)$ reveals the last shuffled position of block a, but not all the shuffle-history positions for a are revealed. The shuffle history for block a is defined as

$$ShuHis(a) = \{(x_1, y_1), (x_2, y_2), \cdots, (x_l, y_l)\},$$

where each (x_j, y_j) is the physical address for storing block a at the j-th ORAM accesses after data shuffling, x_j is the leaf identifier, and y_j is the block height. It is clear that

$$\{(x_{i,a}, y_{i,a})\} \subset ShuHis(a). \tag{12}$$

Theorem 6.1: Let $His(a) = \{(x_{i,a}, y_{i,a})\}$ be the partial shuffle history for block a in tree T_i. It holds that $His(a)$ is smaller than the full frequency pattern $fp(a)$.

Proof: Given the leakage $His(a)$, an adversary can observe the appearances by testing how many times the last matched block is in the paths, x_1-to-root, x_2-to-root, $\cdots$, and x_{l-1}-to-root. Let the number of appearances be X. If the block exists in only one path, i.e., $X = 1$, the adversary will successfully output the correct access sequence of a. However, if $X > 1$, which denotes that the block has been frequently shuffled, and the adversary can output the correct access sequence of a with a probability of approximately $\frac{1}{X}$. Therefore, $His(a)$ is smaller than the full frequency pattern $fp(a)$.

Theorem 6.1 and the above analysis hold that the frequently-accessed logical address is secure compared to the infrequently-accessed address. Fortunately, in reality, most of the attacks show that even if the full frequency pattern is given, only the frequently-accessed address is vulnerable, and the infrequently-accessed addresses are relatively strong [41]. Therefore, the leakage of the FP/BP-ORAM is practically acceptable in DSE applications.

Theorem 6.2: LL-ORAM is an immediate-update ORAM.

Proof: Consider a data block a' in a position LL-tree. If this block has been matched by the user's token, the block will be mapped into a new $leaf'$-to-root path, where $leaf'$ is a randomly-generated leaf identifier. Therefore, the probability of the block staying still is small, and $|StopHis(a')| << |fp'(a')|$, i.e., the immediate-update ORAM.

6.3 Discussions

Where is the next accessing node for the same address? For ease of understanding, let us consider only one position LL-tree T_i. If a set of x-to-root nodes have been retrieved back from T_i, this means that a node key, which is for the address (a_i, b_i), has been used. The user randomly allocates a new leaf identifier x' for this block (a_i, b_i). The next accessing node for (a_i, b_i) will be put into the y-to-root path, where y is the first overlapped node of the x-to-root path and the x'-to-root path. Since the whole x-to-root path has been rebuilt, x' is still hidden from the cloud, and only x is revealed.

Is the next accessing leaf disclosed? After the eviction, the next accessing leaf value x' is stored in T_i and T_{i-1}. In T_i, x' is stored in the tail part of one LL-tree node, which is encrypted by the RCPA-secure algorithm. Thus, it is secure enough. In T_{i-1}, x' is stored in an EMap that is encrypted by a one-time value. No efficient adversary can break the EMap without the one-time value. Therefore, x' is still hidden now.

Note that the next physical access address for the same logical address is always hidden from the cloud after any query, even if the logical address's shuffle history has been revealed. This feature helps LL-ORAM achieve the claimed forward privacy and backward privacy.

6.4 LL-ORAM Security

Theorem 6.2: Assume the probability of the root stash exceeding S is negligible. Assuming F and G are pseudorandom functions, and there exists an RCPA-secure algorithm, then 1) LL-FP-ORAM is level-1 forward-private (FP-I), 2) LL-BP-ORAM is level-1 backward-private (BP-I), and 3) LL-OB-ORAM is perfectly secure.

Proof: From the above analysis, it holds that all the security components are secure enough. Let us analyze the following two protocols.

Consider the protocol $ObAccess(.)$. Since the algorithm $Compute(.)$ is handled by the client, no shuffle history is revealed. The cloud observes only an array of randomly-chosen leaf identifiers. Thus, LL-OB-ORAM is perfectly secure with only the leakage of the ORAM size and the number of ORAM accesses.

Consider the protocol $Access(.)$. The only security concern is the algorithm $Compute$ since it is handled by the cloud now. The computational result reveals the matched data block's physical address, which is the last shuffled address of the desired block, but no more physical addresses are revealed.

According to Theorem 6.1, the algorithm $Access(.)$ leaks no full frequency pattern. According to Theorem 6.2, LL-ORAM is always immediate-update. Because $His(a)$ cannot be expressed as $fp(a)$, $His(a) \subset ShuHis(a)$, and $ObAccess(.)$ is always data-oblivious, LL-FP-ORAM is level-1 forward-private, and LL-BP-ORAM is level-1 backward-private. Even if a shuffle-history address for block a has been revealed, the next physical address for block a is still hidden from the cloud. Thus, the user can switch LL-ORAM's security among forward privacy, backward privacy, and perfect security at any time.

7 Experimental Evaluations

7.1 Experimental Methodology

The experiments are conducted on a desktop computer running Windows 10 with one Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz processor and 40 GB DDR4 memory. The core encryption of LL-ORAM consists of only pseudorandom computations and symmetric encryption. Blake2b acts as the pseudorandom function, and counting-mode AES acts as symmetric encryption. All experimental testing cases and LL-ORAM procedures are entirely implemented in C++.

In the following experiments, for simplicity, the communication time is ignored. Since there is no accounting for client-server interactions, the APIs $Access(.)$ and $ObAccess(.)$ have almost the same computational complexity. We employ a hash table to hold all the tree nodes since the hash table has the advantage of dynamical memory allocation. To obtain the real performance of LL-ORAM, we use only one thread to perform the experiments.

7.2 Root stash size

The experimental results in Table 1 show that the root-stash size is small. Given an L-height LL-ORAM, we can insert $(2^L - 1)$ real blocks into the ORAM at most. Repeatedly performing reads/writes, we observe how many times the roots exceed size S (in chunks). In an L-height ORAM, the probability of the root-stash size exceeding S is written as $\alpha(S, L, Z)$, where Z is the number of chunks in each tree node. After millions of reads and writes with the worst-case access-pattern sequence $\{1, 2, \cdots, N, 1, 2, \cdots, N, \cdots\}$, $\alpha(4, 15, 3)$, $\alpha(4, 20, 3)$, and $\alpha(4, 25, 3)$ are always zero, which means that no root-stash size is equal to or above 5 in the tests. $\alpha(3, 15, 3)$ is approximately $1.36E^{-5}$, $\alpha(3, 20, 3)$ is approximately $2.11E^{-6}$, and $\alpha(3, 25, 3)$ is approximately $2.66E^{-6}$. The conclusion implies that one of the recommended settings for the LL-ORAM is $S = 4$ and $Z = 3$.

Another recommended setting is ($S = 7$, $Z = 2$). Since Z is directly related to communication bandwidth, we attempt to reduce $Z = 3$ to $Z = 2$. If $Z = 2$ is adopted, $\alpha(7, 10, 2)$, $\alpha(7, 20, 2)$, and $\alpha(7, 25, 2)$ are always zero, yet $\alpha(6, 155, 2)$ equals $4.19E^{-7}$ and $\alpha(6, 20, 2)$ is $1.99E^{-7}$. We can conclude that ($S = 7$, $Z = 2$) is one of the good choices in reality. If $Z = 1$, S should be set to 24 or higher to avoid stash exceedance in our experiments. However, a large S usually means poor access efficiency.

TABLE 1: Probability of stash exceeding

α	$(S, Z) = (3, 3)$	$(S, Z) = (4, 3)$	$(S, Z) = (6, 2)$	$(S, Z) = (7, 2)$
$L = 15$	$1.36E^{-5}$	0	$4.19E^{-7}$	0
$L = 20$	$2.11E^{-6}$	0	$1.99E^{-7}$	0
$L = 25$	$2.66E^{-6}$	0	$2.52E^{-7}$	0

7.3 Block size

The experimental results in Figure 7 demonstrate that 1) LL-ORAM can be applied to dynamic searchable encryption without sacrificing much efficiency, and 2) the bulk insertion efficiency of LL-ORAM can be very high if $B \in [64L, 64L^2]$ is adopted. Assume each data block has r records, and each record is a 64-bit integer to hold a value, an SE file identifier.

Figure 7 shows the relationship between insertion speed (the number of insertion identifiers per second) and block size (in records), where $L = 25$ denotes an ORAM whose capacity is 2^{25}, and so on. If $r = 16$, the insertion speed is not high. If $r \approx L^2$, the batch insertion speed reaches the maximum value. For example, if $L = 20$ and $r = 512$, a 20-height LL-ORAM can insert $3.5E4$ file identifiers per second in bulk insertion mode, assuming each identifier occupies 64 bits. It is easy to imagine that all data transferred in bulk are more efficient than sequential order.

However, the larger the block size is, the more blocks should be encrypted, transferred, and shuffled in each access. This implies that a large block (e.g., $B = 300$ KB when $L = 22$) is not suitable for LL-ORAM. Therefore, we choose $r \in [L, L^2]$ that means $B \in [64L, 64L^2]$ bits.

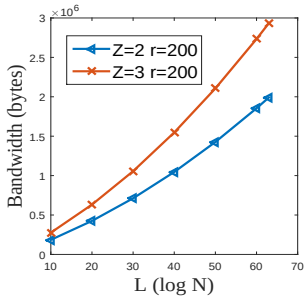

Fig. 5: Communication bandwidth

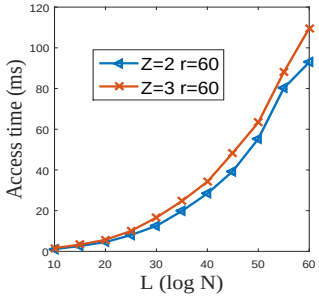

Fig. 6: Access time

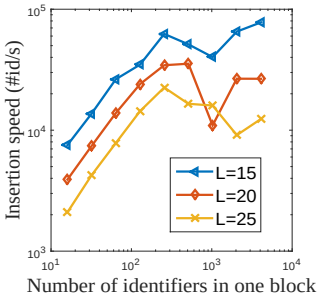

Fig. 7: Bulk insertion speed (in records)

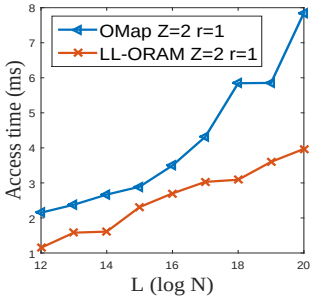

Fig. 8: Compared width with OMap

7.4 Communication Bandwidth

The communication bandwidth of LL-ORAM is the optimal value $O(B \log N)$ bits, only if $B = \Omega(\log^2 N)$ bits. As shown in Figure 5, if $Z = 2$ and $r = 200$ are set, and if the LL-ORAM ideally contains 2^{63} real blocks with 200 records each, to read or write a record, the user will receive 1.9 MB data in the worst case; if $Z = 3$ is adopted, 2.9 MB data will be transferred back to the user since more blocks are in each tree node. More practically, if $L = 30$, which means there are 2^{30} blocks, the user should download 712-KB data to obtain one block that contains 200 records if $Z = 2$ and $r = 200$. We note that this result is acceptable on practical occasions since the user requires making only one request and receiving only one response.

7.5 Access time

Figure 6 shows that the access time is in milliseconds, even if the ORAM capacity reaches 2^{60} blocks. Assuming the memory is sufficient, all the trees exist in the memory. If we set $Z = 2$, $L = 60$, and $r = 60$, to search one block over 2^{60} addressing space, it takes only 92 ms. In a smaller addressing space of $L = 16$, any access consumes 3.58 ms in the worst case with $r = 16$ matched records. The efficiency comes from the compact design: downloading an RLL and uploading an RLL. Since RLL encryption consists of only the Blake2b algorithm and AES that are practically efficient, the RLLs are easy to construct in milliseconds. Building an RLL takes only $O(\log^2 N)$ time. Assuming communication time is not considered, the access time is appropriate $O(\log^2 N)$.

7.6 Compared with OMap

LL-ORAM is still competitive, compared with the state-of-the-art ORAM, OMap (ODS) in [11], which consists of a nonrecursive Path ORAM and a balanced AVL tree. The main defect of OMap is that it suffers from $O(\log N)$-round-trip client-server interactions all the time.

The experimental data in Figure 8 demonstrate that LL-ORAM is more efficient than OMap in terms of access time. In both the experiments, a tree node containing four blocks with each block 64 bits, and all communication times is not captured. The efficiency of LL-ORAM comes from the compact design of the data structures, such as the EMap.

8 Conclusions and Future Work

In this paper, we introduced the forward-/backward-private ORAM definitions and gave LL-ORAM, a new ORAM that consists of a set of switchable protocols. The ORAM has the following advantages: 1) a single-round-trip read/write with forward privacy or backward privacy; 2) near-zero client storage; 3) low computational overhead; 4) $O(\log^2 N)$ read/write time complexity; 5) $O(BN)$ cloud-side storage if $B = \Omega(\log^2 N)$ bits with addressing space $[0, BN]$ in bits; and 6) $O(B \log N)$-bit communication bandwidth if $B = \Omega(\log^2 N)$ bits.

Future work includes the following: 1) algorithmic improvements on FP/BP-ORAMs, 2) the reviews on encrypted Boolean queries and high dimensional queries with an FP/BP-ORAM, 3) multiclient ORAMs and oblivious parallel RAMs, 4) FP/BP-ORAMs in secure hardware and applications, and 5) the studies on FP/BP-ORAM-based DSE schemes.

Acknowledgments

This work is supported by the Research Foundation of Education Bureau of Hunan Province, China, under Grant 19C0085, the National Natural Science Foundation of China (61772454, 61811530332, 61811540410, 61972146), Hunan Provincial Key Research and Development Program (Grant No. 2018GK2055), and Hunan Provincial Natural Science Foundation of China (Grant No. 2020JJ4376).

REFERENCES

[1] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1465–1482. ACM, 2017.

[2] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, pages 707–720, 2016.

[3] Jin Li, Yanyu Huang, Yu Wei, Siyi Lv, Zheli Liu, Changyu Dong, and Wenjing Lou. Searchable symmetric encryption with forward search privacy. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

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

[5] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. A retrospective on Path ORAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

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

[7] Bo Li, Yanyu Huang, Zheli Liu, Jin Li, Zhihong Tian, and Siu-Ming Yiu. Hybridoram: practical oblivious cloud storage with constant bandwidth. *Information Sciences*, 479:651–663, 2019.

[8] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 345–360. ACM, 2019.

[9] Kholoud Saad Al-Saleh and Abdelfettah Belghith. Radix path: A reduced bucket size oram for secure cloud storage. *IEEE Access*, 7:84907–84917, 2019.

[10] Karin Sumongkayothin. M-oram revisited: Security and construction updates. In *International Conference on Information Security Practice and Experience*, pages 521–532. Springer, 2018.

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

[12] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference (CRYPTO)*, pages 563–592. Springer, 2016.

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

[14] Daniel S Roche, Adam Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 178–197. IEEE, 2016.

[15] Syed Kamran Haider and Marten van Dijk. Flat oram: A simplified write-only oblivious ram construction for secure processors. *Cryptography*, 3(1):10, 2019.

[16] Shruti Tople, Yaoqi Jia, and Prateek Saxena. Pro-oram: Constant latency read-only oblivious ram. *IACR Cryptol. ePrint Arch.*, 2018:220, 2018.

[17] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private rams. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 404–434. Springer, 2019.

[18] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server ORAM. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 141–157. Springer, 2018.

[19] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S^3 ORAM: 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 (CCS)*, pages 491–505. ACM, 2017.

[20] Sherman SM Chow, Katharina Fech, Russell WF Lai, and Giulio Malavolta. Multi-client oblivious ram with poly-logarithmic communication. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 160–190. Springer, 2020.

[21] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 660–690. Springer, 2017.

[22] Hamza Omar, Syed Kamran Haider, Ling Ren, Marten Van Dijk, and Omer Khan. Breaking the oblivious-ram bandwidth wall. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 115–122. IEEE, 2018.

[23] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 279–296. IEEE, 2018.

[24] Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 523–535, 2017.

[25] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 218–234. IEEE, 2016.

[26] Zhiqiang Wu, Kenli Li, Keqin Li, and Jin Wang. Fast boolean queries with minimized leakage for encrypted databases in cloud computing. *IEEE Access*, 7:49418–49431, 2019.

[27] Zhiqiang Wu and Kenli Li. Vbtree: forward secure conjunctive queries over encrypted data for cloud computing. *The VLDB Journal*, 28(1):25–46, 2019.

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

[29] Zhangjie Fu, Xinle Wu, Chaowen Guan, Xingming Sun, and Kui Ren. Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(12):2706–2716, 2016.

[30] Jialin Chi, Cheng Hong, Min Zhang, and Zhenfeng Zhang. Fast multi-dimensional range queries on encrypted cloud databases. In *International Conference on Database Systems for Advanced Applications*, pages 559–575. Springer, 2017.

[31] Rui Li, Alex X. Liu, Ann L. Wang, Bezawada Bruhadeshwar, Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. Fast and scalable range query processing with strong privacy protection for cloud computing. *IEEE/ACM Transactions on Networking (TON)*, 24(4):2305–2318, 2016.

[32] Russell WF Lai and Sherman SM Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *International Conference on Applied Cryptography and Network Security*, pages 478–497. Springer, 2017.

[33] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 19–36, 2017.

[34] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized i/o efficiency. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2018.

[35] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1449–1463. ACM, 2017.

[36] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1038–1055. ACM, 2018.

[37] Cong Zuo, Shi-Feng Sun, Joseph K Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption with forward and stronger backward privacy. In *European Symposium on Research in Computer Security (ESORICS)*, pages 283–303. Springer, 2019.

[38] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, page 4. ACM, 2019.

[39] Panagiotis Rizomiliotis and Stefanos Gritzalis. Simple forward and backward private searchable symmetric encryption schemes with constant number of roundtrips. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 141–152. ACM, 2019.

[40] Zichen Gui, Kenneth G Paterson, Sikhar Patranabis, and Bogdan Warinschi. Swissse: System-wide security for searchable symmetric encryption. *IACR Cryptol. ePrint Arch.*, 2020.

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