BI-ORAM: an oblivious bulk-insertion log table

Zhiqiang Wu[1] and Jun Liu[2]

Changsha University of Science and Technology, Hunan, China
[1]cxiaodiao@hnu.edu.cn,[2]jt23-liu@csust.edu.cn

Abstract. The rapid growth of log data necessitates efficient and secure cloud backup methods. While Oblivious RAM (ORAM) can hide data access patterns, existing ORAM designs suffer from high performance overhead during bulk data insertion—a fundamental requirement for logging. Tree-based ORAMs, such as Path ORAM, require multiple rounds of communication and incur a quadratic slowdown when inserting many logs simultaneously. Although newer multi-path ORAMs like OBI (NDSS 2023) improve performance, they demand substantial client-side storage due to a local position map that tracks data locations. This paper presents BI-ORAM, a new ORAM-based log table designed for fast bulk insertion in cloud log backup. BI-ORAM employs a simple date-keyed and hour-indexed log storage structure for efficient subsequent retrieval. Its key innovation is a position-map-free bulk eviction mechanism, which combines a hash function for direct placement of new log blocks with an optimized partition-based eviction strategy. This design enables BI-ORAM to insert r log files in $O(r \log N)$ time with only one round trip of client-server communication, while maintaining $O(1)$ client storage—a significant improvement over prior work. We formally prove that BI-ORAM is statistically secure. Experimental results demonstrate that BI-ORAM achieves substantially faster log insertion than recursive Path ORAM and matches the performance of OBI, but without OBI's large client storage overhead. Hence, BI-ORAM provides a practical and secure solution for privately backing up logs at scale.

Keywords: Oblivious RAM · Oblivious Data Structures · Oblivious bulk insertion.

1 Introduction

1.1 Background and Motivation

Log data, which comprises sequential records of system events, is essential for system monitoring, security auditing, and forensic investigation [14]. It also facilitates predicting market trends and understanding customer needs [23, 30]. In recent years, as systems and cloud services have spread, log data has become among the fastest-growing data types. The long-term retention requirements for such massive log data impose a significant burden on local storage.

The rise of cloud computing has led organizations to store data on cloud servers, taking advantage of their low cost and accessibility [38, 39]. Moving logs to the cloud reduces local storage needs. However, merely encrypting and storing the log data on cloud servers is often insufficient. If data-write access patterns are not protected, even a user's read operation on these logs may leak sensitive information, such as log timestamps, log names, data size, access frequency, and relationships between different data. Such leakage exposes the data to a long line of attacks, including similar-data attacks [13, 20, 24, 25, 27] and known-data attacks [5, 6, 17, 18, 22, 27, 36].

For instance, if logs are stored and accessed in a way that reveals information about them, even the mere retrieval of a single log may leak sensitive details. Consider a scenario where a company suspects that a data breach occurred on a specific date, e.g., October 2, 2025. An analyst then accesses the encrypted file `server_log_20251002.csv` on the cloud server. An attacker who has infiltrated the server (but cannot decrypt the file) can observe this access. The filename itself, or the act of reading data associated with that specific date, reveals the focus of the investigation. Thereby, the attacker learns that the security team is examining activities on October 2nd. This not only confirms the date of the attack but may also allow the attacker to cover their tracks or launch a subsequent attack. This example illustrates how exposed access patterns can compromise security investigations and lead to tangible harm.

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky, is a cryptographic primitive designed to protect such access-pattern leakage. It conceals access patterns by continuously shuffling and re-encrypting data, ensuring that observing the access sequence reveals no information about the actual data being accessed. It is a promising technique for building private log storage systems. We aim to design an oblivious log backup table that fulfills the following user's requirements:

1. **Efficient and privacy-preserving bulk insertion.** Log records are generated continuously in high volume. The system must therefore support efficient upload of these records to the server without leaking any information through write-access patterns.
2. **Privacy-preserving data retrieval.** The system must allow users to retrieve historical logs from any specific time period (e.g., to investigate past events) without revealing to the server any information regarding the query, such as the target time period, the specific records of interest, or the frequency of accesses.

1.2 Limitations of Prior Art

Tree-Based ORAMs. Path ORAM [31, 33] is one of the widely-adopted tree-based ORAM schemes due to its simplicity and efficiency [40]. For example, searching for r data items using a recursive position map (which stores the positions of shuffled items) incurs a time complexity of $O(r \log^2 N)$ and requires $O(r \log N)$ client-server interaction rounds. However, Path ORAM is not optimized for bulk insertion, because it suffers from the large-stash eviction problem

first identified in earlier work and later addressed in [35]. The efficiency of Path ORAM scales poorly with the size of the local stash.

OBI [35], a multi-path tree ORAM, improves bulk-insertion performance by achieving $O(r \log N)$ time complexity with only a single round-trip interaction for evicting r items. Nevertheless, OBI relies on a local position map (implemented as a hash table) stored on the client side. As data volume grows, this position map consumes substantial client storage, becoming a scalability bottleneck. Moreover, every data access (whether for newly inserted or existing blocks) depends on this local map, introducing non-negligible lookup latency during each insertion and thus limiting overall performance.

Hierarchical ORAMs. Hierarchical ORAM constructions, such as PanORAMa [26] and OptORAMa [3], are theoretically optimal, achieving $O(\log N)$ overhead. In practice, however, their large hidden constants often make them inefficient for practical problem sizes, limiting real-world adoption. Recent advancements like FutORAMa [4] and MegaBlocks [2] have significantly optimized these constants, achieving asymptotically optimal performance that is truly practical. A key obstacle for cloud log backup scenarios, however, is their highly interactive protocol design. This interactivity introduces latency and complexity that may be prohibitive. Additionally, thus far, no hierarchical ORAM has been proven statistically secure. This remains a significant theoretical barrier.

1.3 Proposed Approach

In this paper, we propose BI-ORAM, an Oblivious RAM scheme optimized for efficient bulk insertion of log data. The design employs two key techniques to achieve fast bulk log insertion while preserving future query capabilities.

First, we introduce a log indexing and storage scheme. Each log file is assigned a unique, structured key in the format of `YYYYMMDD-NN`, where the date component facilitates time-based grouping, and a daily sequential number `(NN)` ensures uniqueness. Concurrently, we maintain a separate, compact index table that maps coarse time intervals (e.g., hourly slots) to the set of log keys generated within them, enabling efficient temporal range queries.

Second, we adapt and simplify the bulk eviction mechanism of OBI [35] for our setting. A major innovation is the complete elimination of the client-side position map, achieved through a two-step process. For bulk insertion: The initial storage position for each log file is computed using a pseudo-random function (PRF) seeded with the file's unique key, thereby obviating the need for a persistent position map during insertion. For file retrieval: We leverage the standard recursive Path ORAM to act as the position map of BI-ORAM. If a position mapping for the queried log is not found in the recursive Path ORAM, the client falls back to computing its initial position using the PRF (as during insertion).

1.4 Our Contributions

1. We propose BI-ORAM, an oblivious log table designed for bulk insertion. It achieves highly efficient bulk insertion while maintaining minimal client-side storage overhead.
2. We introduce an enhanced bulk-insertion algorithm that significantly reduces the time complexity for inserting new data. Our method requires only $O(r)$ time to locate the positions of r new items and only $O(r \log N)$ time to complete the bulk insertion, representing a significant improvement over prior schemes.
3. We formally prove that BI-ORAM is statistically secure.

2 Notations and Definitions

2.1 Threat Model

We consider a system consisting of two parties: a fully trusted client and an honest-but-curious server. The client's local execution is considered trusted and is not observable by any adversary. The server faithfully follows the prescribed protocol. During protocol execution, an adversary continuously monitors the server's activities and may attempt to deduce sensitive information from the interaction protocol. Our security goal is to protect data access patterns from being inferred by such an adversary.

2.2 Security Model

We adopt the statistical security notation for an ORAM scheme. Let Π be an ORAM that translates client-side logical memory access sequences into server-side physical requests. Let N be the ORAM capacity.

Definition 1 (Statistically secure ORAM). *An ORAM Π is* statistically secure *if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that for any polynomial-length sequence of logical memory requests $\mathcal{R}$ (where $\mathcal{L}(\mathcal{R}) = (N, |\mathcal{R}|)$), the following ensembles are statistically indistinguishable:*

$$\Pi(\mathcal{R}) \stackrel{s}{\equiv} \mathcal{S}(\mathcal{L}(\mathcal{R}))$$

where $\stackrel{s}{\equiv}$ denotes statistical indistinguishability (i.e., the statistical distance between the distributions is negligible).

This security definition guarantees that:

- The server's view (the sequence of physical memory accesses) can be perfectly simulated given only the ORAM capacity and the total number of logical operations.
- No computationally unbounded adversary can distinguish between the real ORAM access patterns and the simulated patterns.
- The security holds regardless of the actual data values or access patterns in the logical request sequence $\mathcal{R}$.

2.3 Path ORAM

Path ORAM [33, 31] consists of three main components: a binary tree stored on the server, a recursive position map (also stored remotely), and a local stash. Each node of the binary tree, called a bucket, contains multiple encrypted data blocks. Every block is assigned to a random leaf and must reside in one of the buckets along the path from the root to that assigned leaf.

To access a data block, the client first queries the recursive position map to obtain the block's current leaf identifier and reassigns it a new random leaf identifier. Then, the client reads all buckets along the path from the root to that leaf from the server, decrypts them, and loads the contained blocks into a local stash. After reading or updating the target block in the stash, the client re-encrypts the blocks, reorders them obliviously, and writes them back into the same path. This path-based access pattern effectively conceals which specific block was accessed. The time complexity of a single access in (recursive) Path ORAM is $O(\log^2 N)$, where block size is $\Theta(\log N)$ bits.

The non-recursive Path ORAM stores the position map at the client. This reduces the access time to $O(\log N)$ but introduces significant client-side storage overhead for maintaining the map.

2.4 The Problem of Bulk Insertion in Path ORAM

Path ORAM [33] suffers from the large-stash eviction problem first studied in OBI [35]. Path ORAM has two approaches to evicting a large stash. One is to write all data into the client's stash and then perform a bulk eviction across multiple paths in the ORAM tree. However, this requires the eviction algorithm to perform a linear scan of the entire stash for each path, incurring significant computational overhead, especially when the stash size is large. The alternative is to follow the standard Path ORAM protocol, evicting data in many small, incremental batches. While computationally lighter per batch, this method necessitates a high number of communication rounds with the remote server, leading to substantial latency and bandwidth costs.

2.5 OBI

OBI [35] is a multi-path ORAM architecture built on three core components: an encrypted binary tree stored on the cloud server, a local stash, and a local hash table that acts as a client-side position map. In each access, OBI reads multiple tree paths and relies on a dedicated multi-path eviction algorithm. A key design rule governs eviction: a data item can only be moved to a tree node that lies on the intersection of its own assigned leaf path and the current eviction path. OBI is secure under the strong forward and backward security model.

To overcome the inefficiency of bulk operations, OBI introduces two novel eviction algorithms: the *k-Nearest Neighbor Eviction Algorithm (KNNEA)* and the *Partition-Based Eviction Algorithm (PBEA)*.

KNNEA is optimized for small to moderate datasets. It first sorts stash entries by their leaf identifiers. During eviction, rather than scanning the entire stash, it examines only the k entries whose leaf is nearest to the leaf of the target node, where $k = 2Z + 1$ and Z is the bucket capacity. The algorithm fills tree nodes from bottom to top along each accessed path, iteratively placing the most suitable blocks until all nodes are processed.

PBEA targets large-scale workloads. It partitions the leaf-identifier space into intervals of fixed size and distributes stash entries into corresponding partitions. Each non-empty partition is then processed independently using a variant of KNNEA. Any remaining stash entries are finally evicted via a standard KNNEA procedure. This two-stage design reduces the eviction time complexity from $O(r^2)$ to $O(r \log N)$, where r is the number of accessed paths.

Both KNNEA and PBEA avoid linear scans of the stash and substantially improve the throughput of bulk-insertion operations. In our scheme, we adopt PBEA as the underlying eviction engine and refer to its interface as OBI.PBEA(). At the same time, the $OBI.Access()$ interface is used to represent a single data read and write operation.

3 Design of BI-ORAM

3.1 Log-file Structures

key	value
key_0: '20251127: 0'	($time_0$, index)
key_1: '20251127: 1'	($time_1$, ⋯)
key_2: '20251127: 2'	($time_2$, ⋯)
...	...

index

'1, 1, 1000'
'2, 1001, 1601'
'3, 1602, 9900'
...

Fig. 1. Data structures of log files.

BI-ORAM employs the log-file structures illustrated in Fig. 1, which consist of two core components: a `(key, value)` pair and a daily `index`. This design addresses the system's key requirements: managing a massive volume of individually small log files, and accommodating highly variable write traffic within short time intervals (e.g., one minute or one hour).

A `(key, value)` pair. Each log file is assigned a globally unique key for retrieval. The key follows the format `'date:number'`, where `date` is in `YYYYMMDD` and `number` is an integer that increments daily from zero without a predefined

upper bound. The key `date:0` is reserved for the day's index. For example, `20251127:0` identifies the index for November 27, 2025, while `20251127:657` uniquely points to that day's 657th log. The corresponding `value` field stores the log's actual content.

Daily `index`. This is a fixed-size array that abstracts the day's log activity. Each slot records the range of log sequence numbers generated within a specific hour. For instance, if logs 1001 through 1601 were produced in the second hour, the entry would be `index[2] = '2,1001,1601'`. This daily index is itself treated as a log file. Upon uploading the final batch of logs for the 24th hour, the daily index is also persisted to the cloud.

3.2 BI-ORAM Structures

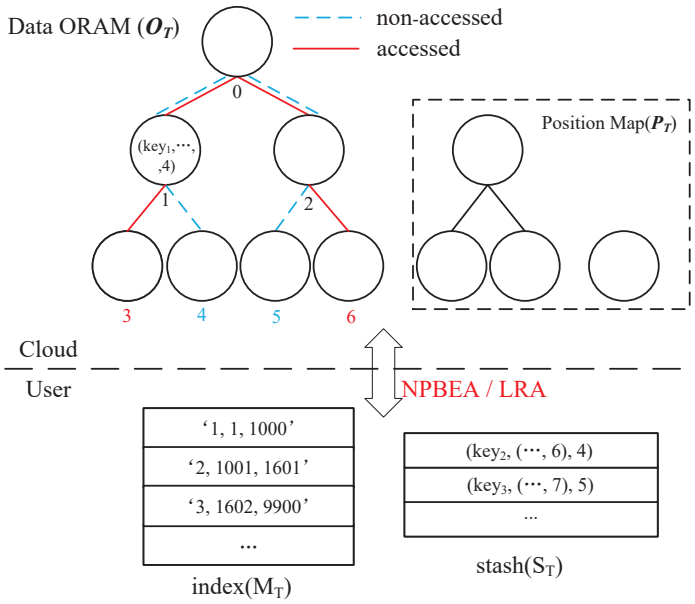

Fig. 2. A BI-ORAM overview.

The BI-ORAM system consists of four parts: a data ORAM (O_T) and a position ORAM (P_T) on the cloud server; a local stash (S_T) and a daily index (M_T) on the client, as shown in Figure 2.

On the cloud side, the data ORAM (O_T), built upon the OBI system, is a full binary tree with a height of $L = \log N$, where N is the ORAM capacity. Each tree node has Z slots. Each slot holds a block with three fields: $(key, value, leaf)$. Here, key and $value$ represent the actual log data, and $leaf$ is a number that

points to a leaf node in the tree. This $leaf$ number denotes that this block belongs to the path from that leaf up to the root. The entire tree is stored on the cloud in an encrypted form. The position ORAM (P_T), built upon a small, recursive Path ORAM, stores the position information of shuffled log files. P_T acts as a special position map of O_T. Since P_T only stores the key-to-leaf mappings, it is much smaller than the data tree O_T.

On the client side, two components are maintained. Local Stash (S_T): A hash table that temporarily holds triples pending processing or recently accessed, structured as $S_T[key] = (key, value, leaf)$. In-Memory index ($M_T$): The index structure for the current day (an array) resides in client memory. At the end of the day, it is written to the cloud as a triple that can be retrieved by the keyword $'date : 0'$. The specific implementation is shown in Section 3.1.

We make an important improvement: when inserting logs in bulk, we do not read the position map (P_T). Instead, an initial $leaf$ for each new log $(key, value)$ is directly computed by using $h(key)$. Specifically, $h(key)$ is implemented as:

$$h(key) = (2^{L-1} - 1) + (F_{\mathbb{K}}(key) \% 2^{L-1}),$$

where F is a keyed collision-resistant hash function, and $\mathbb{K}$ is the user's secret key. This formulation ensures that each key deterministically maps to a leaf identifier within the valid range of the ORAM tree.

During the batch insertion of new data, the initial leaf identifier for each new data item is computed using the hash function h(key). After that, an eviction operation is performed along a randomly selected path. Take Figure 2 as an example. Assume the cloud storage already holds some data, such as $(key_1, \ldots, 4)$. Now consider a set of new data to be inserted by the client: $\{(key_2, (\ldots, 6)), (key_3, (\ldots, 7)), \ldots\}$. We use the hash function $h()$ to generate a leaf identifier for each data item, which determines the path where it should be stored after shuffling. For instance, two of these data items can be represented as triples: $t_2 = (key_2, (\ldots, 6), h(key_2) = 4)$ and $t_3 = (key_3, (\ldots, 7), h(key_3) = 5)$, where 4 and 5 are the leaf identifiers for t_2 and t_3, respectively. Suppose paths 3 and 6 are randomly chosen as the eviction paths. According to the ORAM eviction rule, t_2 and t_3 can only be evicted to the currently accessed red path or the as-yet-unaccessed blue path, and these two paths intersect at certain nodes in the tree. For t_2, for example, it can only be evicted to node 1 or node 0 (only when node 1 is full can it be further evicted to node 0).

3.3 Access algorithms

We propose two algorithms: $NPBEA$ and LRA. The first is for bulk insertion, and the second is for data retrieval. Table 1 lists the parameters of $NPBEA$, and Algorithm 1 shows its specific workflow.

Position-map-free bulk insertion. It is an oblivious bulk insertion technique that does not require a position map during insertion. Each data block is assigned an initial position by a PRF, eliminating the need to read the recursive Path ORAM.

Table 1. Parameters of NPBEA

Notation	Meaning
key	keyword index in triplets
$value$	data to be written to the cloud
$leaf$	a leaf identifier
t	$(key, value, leaf)$, a triplet
$\boldsymbol{T_D}$	$\{(key_1, value_1), \cdots, (key_r, value_r)\}$, a set of data pairs to be inserted
$\boldsymbol{S_T}$	the triplets stored in the local stash
$\boldsymbol{I}$	$\{leaf_1, \cdots, leaf_r\}$, a set of leaf identifiers
$\boldsymbol{T_N}$	a set of tree nodes stored in a hash table
$downPaths(\boldsymbol{I})$	download path nodes corresponding to $\boldsymbol{I}$ from the cloud
$uploadPaths(\boldsymbol{T_N}, \boldsymbol{I})$	encrypt $\boldsymbol{T_N}$ and replace the cloud paths $\boldsymbol{I}$

Algorithm 1: Non-Position-map Partition-Based Eviction Algorithm (NPBEA)

```
1  NPBEA(T_D):
2  I = {};
3  for d ∈ T_D do
4      leaf ← h(d.key);
5      t ← (d.key, d.value, leaf);
6      S_T ← S_T ∪ t;
7      I ← I ∪ RandomLeaf();
8  S_T ← S_T ∪ downPaths(I);
9  T_N ← OBI.PBEA(S_T, I);
10 uploadPaths(T_N, I);
```

NPBEA takes as input a set of log data pairs, denoted as $\boldsymbol{T_D}$. The process consists of four steps: (1). Let $r = |\boldsymbol{T_D}|$. It reads r random paths from the cloud and writes them into the stash. Let $\boldsymbol{I} = \{leaf_1, leaf_2, \ldots, leaf_r\}$ denote the identifiers of these paths. (2). Each key key_i is hashed to obtain its initial leaf identifier $h(key_i)$. The data pairs are then converted into a set of triples: $\{(key_1, value_1, h(key_1)), (key_2, value_2, h(key_2)), \cdots, (key_r, value_r, h(key_r))\}$, which are stored in $\boldsymbol{S_T}$. (3). Local shuffling and eviction: The PBEA algorithm from the OBI library is invoked to obliviously shuffle all triples in $\boldsymbol{S_T}$ (including both newly inserted data and previously downloaded data). This ensures that each key remains unique on its path and arranges the triples into a temporary tree node structure $\boldsymbol{T_N}$. (4). Uploading and replacing the paths with $uploadPaths()$: The shuffled and re-encrypted $\boldsymbol{T_N}$ is written back to the cloud, overwriting the original r paths ($\boldsymbol{I}$). This approach achieves a time complexity of $O(r \log N)$ with a single round trip, outperforming traditional ORAM schemes (such as Path ORAM) that require $O(r \log^2 N)$ time and $O(r \log N)$ round trips.

Log retrieval algorithm. LRA is designed to retrieve an existing log file, and its detailed workflow is illustrated in Algorithm 2.

Algorithm 2: Log Retrieval Algorithm (LRA)

```
1  LRA(date):
2  key_0 ←' date : 0';
3  leaf_0, newleaf_0 ← PositionMap.Access(key_0);
4  if leaf_0 = null then
5      leaf_0 ← h(key_0);
6  t ← OBI.Access(key_0, leaf_0, newleaf_0);
7  M_T ← t.value;
8  num ← userChoose(M_T);
9  key_1 ←' date : num';
10 leaf_1, newleaf_1 ← PositionMap.Access(key_1);
11 if leaf_1 = null then
12     leaf_1 ← h(key_1);
13 t ← OBI.Access(key_1, leaf_1, newleaf_1);
14 value* ← t.value;
15 return value*;
```

For log retrieval, *LRA* takes a specific *date* as input and returns the corresponding log file based on the user's selection. It proceeds through the following steps: (1) Obtain the daily index by constructing the index key as key_0 = 'date:0'; then retrieve and update the position of key_0 via $PositionMap.Access()$. Let $leaf_0$ be the current position and $newleaf_0$ be the next position, which is a new random value. (2) Determine whether $leaf_0$ is null. (Note: The value of $leaf_0$ can only be determined after decryption on the client side, and the server cannot know its content.) If $leaf_0$ is null, set $leaf_0 = h(key_0)$. (3) Execute $OBI.Access()$ to access the target triplet t corresponding to key_0, and retrieve the daily index structure M_T. While accessing with $leaf_0$, $OBI.Access()$ uses the leaf identifier $newleaf_0$ to shuffle the triplet t and store it back to the cloud. (4) Call userChoose(). This function is executed by the trusted client. Given the input M_T, i.e., the index for that day, the user determines which log to retrieve. Here, we denote the number as *num*. (5) Based on *date* and *num*, construct the index $key_1 = date : num$. Then, retrieve and update the position of key_1 via $PositionMap.Access()$. Let $leaf_1$ be the current position and $newleaf_1$ be the next position, which is a new random value. (6) If $leaf_1$ is null, set $leaf_1 = h(key_1)$. Finally, use the read method of OBI to retrieve the log file and return it to the user.

Note that during the execution of the LRA algorithm, the leaf identifier corresponding to each *key* is a random value and changes with each access (it is not fixed). $PositionMap.Access()$ not only reads a leaf identifier based on the *key* but also assigns a new random value for the next read of the *key*. The position map (P_T) always stores these varying positions.

3.4 Interaction Process

BI-ORAM supports oblivious bulk insertion and data retrieval. The overall access algorithm is shown in Algorithm 3.

The access algorithm takes as input an operation type *op* (either read or write), a *date*, and—for write operations—a set of data pairs to be inserted ($\boldsymbol{T_D}$). Based on the value of *op*, the algorithm invokes either $LRA()$ or $NPBEA()$. For a read operation, it calls $LRA()$ with the given *date* and returns the corresponding log file to the user. For a batch insert operation, it passes the data pairs $\boldsymbol{T_D}$ to $NPBEA()$, which downloads and flushes the data before writing it back to the cloud.

Algorithm 3: BI-ORAM Access Algorithm

```
1 Access(op, date, T_D):
2 if op = 'read' then
3     return LRA(date);
4 else
5     NPBEA(T_D);
```

During the above read and write processes, BI-ORAM adheres to the *key* uniqueness principle. For any log file stored in the tree or stash, we ensure that the log has a unique *key*, because *key* is generated by the client through the current date and number when it is first inserted, and the same number on the same date will be generated only once. At the same time, in any subsequent interaction protocol, the correspondence between the log file and its *key* will not be changed.

4 Security analysis

Claim 1: Assume the existence of a secure pseudo-random function and a secure randomized private-key symmetric encryption scheme. BI-ORAM is statistically secure with leakage limited to the number of accessed leaf nodes.

Proof: Consider the BI-ORAM protocol Π, which consists of two operations: bulk insertion and log entry read. Π generates an array of requests $\mathcal{R}$. We now construct a simulator $\mathcal{S}$, which is given only $\mathcal{L}(\mathcal{R})$ to adaptively simulate Π's requests.

In the initial setup stage, based on $\mathcal{L}(\mathcal{R})$, $\mathcal{S}$ creates two simulated ORAM structures OBI^* and $PositionMap^*$, where each is filled with random values. In this step, OBI and OBI^* are indistinguishable, and $PositionMap$ and $PositionMap^*$ are also indistinguishable.

In the query stage, $\mathcal{S}$ also adaptively simulates Π's query. If a request is to access $PositionMap$, $\mathcal{S}$ generates a simulated request to $PositionMap^*$, and

fills the accessed paths with random values. Because $PositionMap$ (i.e., Path ORAM) is statistically secure, $PositionMap$ and $PositionMap^*$ are indistinguishable. If a request is to access the data ORAM (i.e., OBI), there are two types of requests to access OBI: bulk-insertion requests and log-read requests. A bulk-insertion request accesses a set of paths of OBI, and a log-read request (single-point) accesses only one path of OBI. Assume the accessed paths are $(p_1, p_2, \cdots, p_r)$. The simulator creates an array of random paths $(p_1^*, p_2^*, \cdots, p_r^*)$ of OBI^* to simulate Π's requests. Notice that, in a bulk-insertion request, for each path p_i ($i \in [0, r-1]$), its leaf identifier is evaluated by $p_i.leaf = h(key)$. Since $h(\cdot)$ is a secure pseudo-random function, key is unique, and $h(key)$ is used only once, $p_i.leaf$ and $p_i^*.leaf$ are indistinguishable. In a log-read request, the leaf identifier is $p_i.leaf^1$, which is generated by Path ORAM. Thus $p_i.leaf^1$ and $p_i^*.leaf$ are also indistinguishable. Recall that $p_i.content$, the content of the tree node, is encrypted by a secure randomized private-key encryption algorithm (such as counter-mode AES); thus $p_i.content$ and $p_i^*.content$ are also indistinguishable. Therefore, $(p_1, p_2, \cdots, p_r)$ and $(p_1^*, p_2^*, \cdots, p_r^*)$ are indistinguishable.

From the above deduction, we conclude that BI-ORAM is statistically secure with leakage limited to the number of accessed leaf nodes.

With the above effort, we build a secure and efficient log-backup table, which thoroughly protects user's query privacy, including log timestamps, log names, data size, access frequency, and more. Both similar-data attacks [13, 20, 24, 25, 27] and known-data attacks [5, 6, 17, 18, 22, 27, 36] rely on access-pattern leakage. BI-ORAM, however, eliminates such leakage by concealing all user access patterns. Additionally, during bulk insertions, BI-ORAM can read extra fake paths to reduce the leakage of the volume size of the inserted data. As a result, BI-ORAM can effectively thwart these attacks with high probability.

5 Performance Analysis

5.1 Experimental Methodologies

The experiments were conducted on a desktop computer running Windows 10, equipped with an Intel(R) Core(TM) i5-12490F CPU and 32 GB of DDR4 memory. Blake2b was employed as the pseudo-random function in the experiments, and AES in counter mode was adopted as the RCPA-secure encryption algorithm. Both the scheme and test cases were implemented using the C++20 programming language.

We implemented the log table storage, backup, and query system BI-ORAM, and utilized the Path ORAM method to implement log backup functionality in order to compare the efficiency of our solution with that of the Path ORAM based approach.

During the bulk insertion process, it is assumed that the number of new logs inserted into the system in batches per day is M, L denotes the height of the tree on the server, Z represents the number of logs each tree node can store, and r is the number of paths for bulk eviction. In the experiments, M is set to a

relatively large value to simulate the usage scenario of a log server. It is assumed that the entire ORAM can be fully loaded into memory. All communication time is ignored in the experiments. For each experiment, a separate thread was created to evaluate performance. The storage overhead does not include the number of triplets temporarily retrieved during each access. The bulk insertion operation does not consider cases where the ORAM is full, as some space is always reserved for insertion.

In our experiments, to ensure consistency of the controlled variables, BI-ORAM, Path ORAM, and OBI all employ a cloud-based recursive position map by default, unless otherwise specified. All parameter configurations mentioned refer specifically to the Data ORAM (O_T), while the recursive position map maintains the same default parameters throughout.

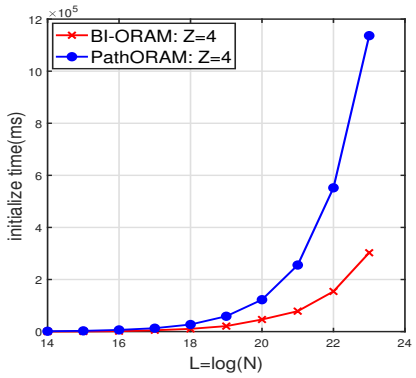

Fig. 3. Comparison of initialization time between BI-ORAM and Path ORAM.

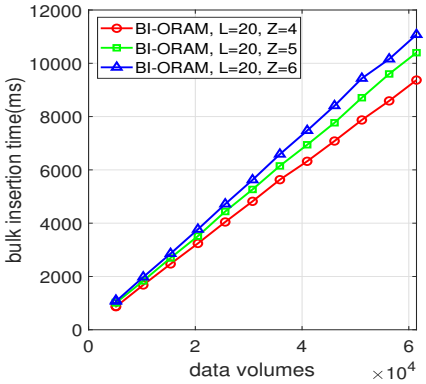

Fig. 4. Comparison of bulk insertion time with different bucket sizes (Z).

5.2 Parameter Tuning Experiments

We first evaluate the system initialization performance of BI-ORAM. As illustrated in Fig. 3, BI-ORAM is compared with a Path ORAM based scheme under the same parameter $Z = 4$ and a range of ORAM tree heights, while inserting an identical volume of initial data. The results confirm that the initialization efficiency of BI-ORAM is approximately four times higher than that of Path ORAM.

Fig. 4 analyzes the impact of the bucket size Z on insertion performance while keeping the tree height and data volume fixed. The results show that a larger Z leads to lower insertion efficiency, which can be attributed to the increased overhead of evicting invalid data. For consistency in subsequent experiments, we set $Z = 4$.

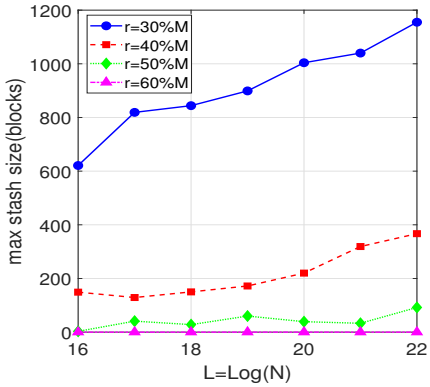

Fig. 5. Comparison of insertion time with different numbers of eviction paths.

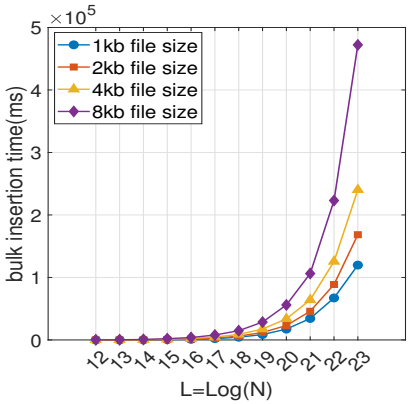

Fig. 6. Comparison of insertion time with different data block sizes.

Fig. 5 demonstrates the relationship between the number of eviction paths r and the remaining data size in the stash after eviction. The experiment is conducted with different tree heights L, a fixed eviction data volume M, and an initial tree occupancy $B = 1/L$. We set $Z = 4$ and vary r across 30%, 40%, 50%, and 60% of M. The results indicate that when $r = 50\%M$, the residual data in the stash is already minimal. At $r = 60\%M$, the stash is nearly empty. Therefore, in practice, we set the number of eviction paths to at least 60% of the eviction data volume. In later experiments, to ensure sufficient eviction and balanced data distribution in cloud storage, we simply set $r = M$.

Fig. 6 evaluates the insertion efficiency of BI-ORAM under different block sizes while fixing $Z = 4$. For each test, data is inserted until the tree reaches its maximum capacity $1/L$. It can be observed that the insertion efficiency decreases as the block size increases. This suggests that BI-ORAM is more suitable for scenarios involving a large number of small files, which aligns well with the characteristics of log storage applications.

5.3 Comparative Experiments

Table 2. Bulk-insertion performance comparison for M items

Schemes	Round trips	Client size
Non-recursive Path ORAM	$O(M)$	$O(M)$
Path ORAM	$O(M \cdot \log N)$	$O(1)$
OBI	1	$O(M)$
BI-ORAM	1	$O(1)$

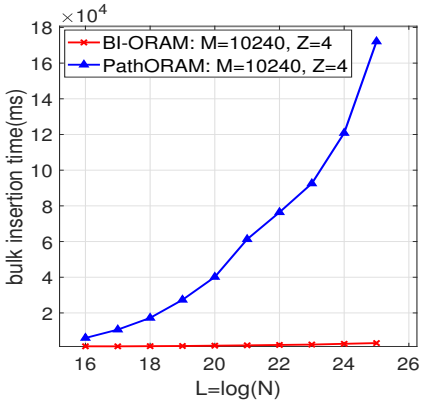

Fig. 7. Comparison of bulk insertion time under different heights.

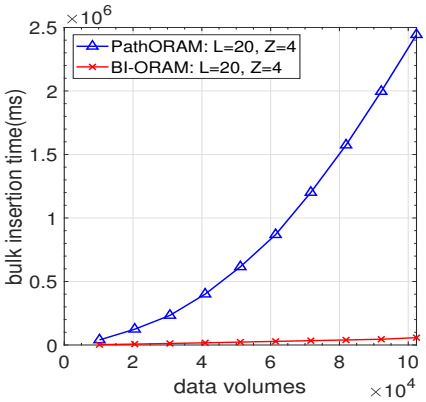

Fig. 8. Comparison of bulk insertion time under different data volumes.

Table 2 presents a parameter comparison among different schemes when inserting M data blocks (with the number of read paths $r = M$). The non-recursive Path ORAM requires $O(M)$ rounds of interaction with the cloud and incurs $O(M)$ client-side storage overhead for the local position map. Although the recursive Path ORAM reduces local storage, it introduces even more interaction rounds. The OBI scheme requires only one round of cloud interaction for bulk insertion but still maintains an $O(M)$ local position map overhead. In contrast, our BI-ORAM scheme requires only one cloud interaction for inserting M data blocks and achieves $O(1)$ client-side storage overhead.

To ensure a fair comparison, all Path ORAM results refer to its recursive variant, as its recursive position map is also stored in the cloud. Fig. 7 compares the insertion performance under different tree heights while inserting the same amount of data $M = 10240$. Fig. 8 shows the performance under a fixed tree height $L = 20$ and $Z = 4$ while varying the inserted data volume M. In both experiments, the load factor of the tree before insertion is set to $\beta = 1/L$. Each log file inserted is 1kb in size.

The results from Fig. 7 and Fig. 8 indicate that, under identical parameters, BI-ORAM significantly outperforms Path ORAM in bulk insertion efficiency. The performance gap widens as the data volume increases. For instance, at a data volume of 10^4 blocks, BI-ORAM achieves an efficiency approximately 10^2 times higher than that of Path ORAM. This is because BI-ORAM has a time complexity of $O(r \log N)$ for bulk insertion, whereas Path ORAM requires $O(r^2 \log N)$. Moreover, BI-ORAM requires only a single round of cloud interaction regardless of the data volume, which leads to an increasingly substantial time advantage as r grows.

Fig. 9 compares the efficiency of BI-ORAM and OBI when performing batch insertion of the same amount of data. To ensure a fair comparison and avoid the influence of other factors on batch insertion performance, we modified OBI by

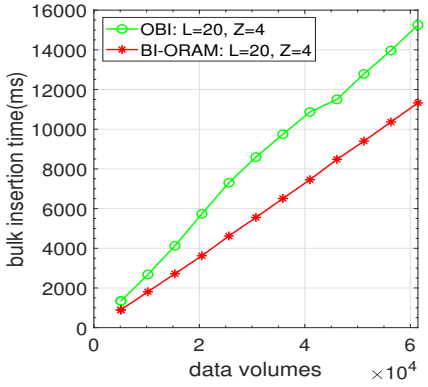

Fig. 9. Comparison of bulk insertion time between BI-ORAM and OBI.

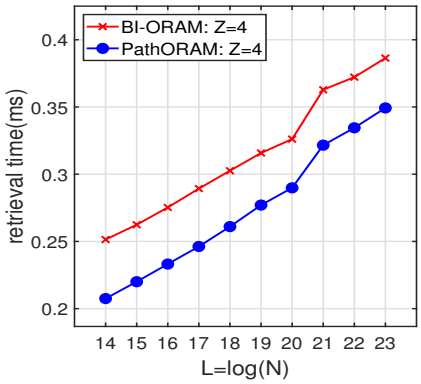

Fig. 10. Comparison of retrieval time between BI-ORAM and Path ORAM.

replacing its local position map with a recursive position map. Additionally, the Data ORAM (O_T) in both schemes was configured with the same parameters L and Z. The results indicate that BI-ORAM exhibits slightly faster insertion efficiency than OBI. This is because OBI must access the recursive position map during bulk insertion, and the overhead of reading the recursive position map increases with the data volume. In contrast, BI-ORAM does not require recursive position map lookups during bulk insertion; it derives block locations through constant-time computations.

Finally, Fig. 10 compares the time consumption for BI-ORAM and Path ORAM to retrieve individual data under the same tree configuration. The results indicate that BI-ORAM exhibits slightly lower retrieval efficiency than Path ORAM. This is primarily because BI-ORAM requires an additional index read operation per query, and locating a block inserted in the first bulk entails extra computation. Nevertheless, BI-ORAM supports occasional specified log file retrieval with an additional time cost of only about $0.1ms$.

6 Related Work

Since its seminal introduction by Goldreich and Ostrovsky [15, 16], Oblivious RAM (ORAM) has evolved from a theoretical construct into a practical cryptographic primitive for concealing data access patterns. Early hierarchical solutions achieved poly-logarithmic overhead but were hindered by large hidden constants. A significant shift occurred with the emergence of tree-based ORAMs, most notably Path ORAM [32, 31, 33], which offered a simpler, more practical construction with $O(\log N)$ bandwidth overhead. Subsequent refinements have focused on reducing constants and client storage [28], solidifying tree-based ORAMs as a mainstream approach.

To meet diverse performance and security requirements, research has branched into several directions. Hierarchical ORAMs, such as PanORAMa [26] and OptORAMa [3], achieve asymptotic optimality, yet their practical deployment is often limited by large hidden constants. Recent work like FutORAMa [4] improves concrete efficiency but relies on computational security assumptions. Asharov [2] closes the theoretical gap in the asymmetric setting by constructing an optimal hierarchical ORAM with $O(\log N/\log\log N)$ I/O overhead and demonstrates significant practical improvements over prior schemes. Distributed ORAMs, exemplified by Duoram [34], employ multi-party computation to distribute trust and enhance bandwidth efficiency. Specialized ORAM schemes have also been developed for various applications, including searchable encryption [21, 35], file-sharing systems [11], and secure computation [10].

A notable research thrust focuses on optimizing bulk and range operations. Schemes such as rORAM [7] and multi-range ORAM [9] improve the efficiency of range queries. For bulk processing, Fork Path [43] and Hitchhiker [44] aggregate requests to eliminate redundant memory accesses. BULKOR [19] improves the theoretical complexity of oblivious bulk loading of a database from $O(N\log^3 N)$ to $O(N\log^2 N)$, but it only supports processing the entire database. Concurrently, forward and backward security has become crucial for searchable encryption, with attacks [37, 41] underscoring the need for robust security models.

Recent advances continue to expand the frontiers of ORAM. V-ORAM [40] proposes an adaptive framework for dynamic workloads, while H2O2RAM [42] introduces a hierarchical doubly oblivious design for high performance. Efforts to support concurrent access [12, 8, 29] and new computation models [1] further broaden the applicability of ORAM in practical systems.

7 Conclusion

In this paper, we propose BI-ORAM, an oblivious bulk-insertion log table, providing a light-weight and cheap approach to protecting user's essential log data. By leveraging the inherent characteristics of log files, BI-ORAM enables oblivious bulk insertion without requiring access to a recursive position map, while also providing query mechanisms based on date and time. BI-ORAM can protect privacy during data reading and writing operations. Extensive experimental evaluations demonstrate its advantages in bulk insertion performance compared to existing schemes. Future work includes fully utilizing the characteristics of BI-ORAM and extending it to broader application domains.

References

1. Appan, A., Heath, D., Ren, L.: Oblivious single access machines-a new model for oblivious computation. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 3080–3094 (2024)
2. Asharov, G., Eiluz, E., Komargodski, I., Lin, W.K.: Megablocks: Breaking the logarithmic i/o-overhead barrier for oblivious ram. In: Proceedings of the 2025

ACM SIGSAC Conference on Computer and Communications Security. pp. 4692–4706 (2025)
3. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: Optorama: optimal oblivious ram. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 403–432. Springer (2020)
4. Asharov, G., Komargodski, I., Michelson, Y.: Futorama: a concretely efficient hierarchical oblivious ram. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3313–3327 (2023)
5. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. Cryptology ePrint Archive (2019)
6. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. pp. 668–679 (2015)
7. Chakraborti, A., Aviv, A.J., Choi, S.G., Mayberry, T., Roche, D.S., Sion, R.: roram: Efficient range oram with o (log2 n) locality. In: NDSS (2019)
8. Chakraborti, A., Sion, R.: Concuroram: High-throughput stateless parallel multiclient oram. arXiv preprint arXiv:1811.04366 (2018)
9. Che, Y., Wang, R.: Multi-range supported oblivious ram for efficient block data retrieval. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 369–382. IEEE (2020)
10. Chen, H., Chillotti, I., Ren, L.: Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 345–360 (2019)
11. Chen, W., Popa, R.A.: Metal: A metadata-hiding file-sharing system. In: NDSS Symposium 2020 (2020)
12. Cheng, W., Sang, D., Zeng, L., Wang, Y., Brinkmann, A.: Tianji: Securing a practical asynchronous multi-user oram. IEEE Transactions on Dependable and Secure Computing **20**(6), 5143–5155 (2023)
13. Damie, M., Hahn, F., Peter, A.: A highly accurate $Query - Recovery$ attack against searchable encryption using $Non - Indexed$ documents. In: 30th USENIX security symposium (USENIX Security 21). pp. 143–160 (2021)
14. Deli, M.S.M., Ismail, S.A., Kama, N., Yusop, O.M., Azmi, A., Yahya, Y.: Malware log files for internet investigation using hadoop: A review. In: 2017 IEEE Conference on Big Data and Analytics (ICBDA). pp. 87–92. IEEE (2017)
15. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 182–194 (1987)
16. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3), 431–473 (1996)
17. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Ndss. vol. 20, p. 12 (2012)
18. Lambregts, S., Chen, H., Ning, J., Liang, K.: Val: Volume and access pattern leakage-abuse attack with leaked documents. In: European symposium on research in computer security. pp. 653–676. Springer (2022)
19. Li, X., Luo, Y., Gao, M.: Bulkor: Enabling bulk loading for path oram. 2024 IEEE Symposium on Security and Privacy (S&P) pp. 4258–4276 (2024)
20. Liu, C., Zhu, L., Wang, M., Tan, Y.a.: Search pattern leakage in searchable encryption: Attacks and new construction. Information Sciences **265**, 176–188 (2014)
21. Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: An efficient oblivious search index. In: 2018 IEEE symposium on security and privacy (SP). pp. 279–296. IEEE (2018)

22. Ning, J., Huang, X., Poh, G.S., Yuan, J., Li, Y., Weng, J., Deng, R.H.: Leap: leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. pp. 2307–2320 (2021)
23. Oliner, A., Stearley, J.: What supercomputers say: A study of five system logs. In: 37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07). pp. 575–584. IEEE (2007)
24. Oya, S., Kerschbaum, F.: Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In: 30th USENIX security symposium (USENIX Security 21). pp. 127–142 (2021)
25. Oya, S., Kerschbaum, F.: $IHOP$: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2407–2424 (2022)
26. Patel, S., Persiano, G., Raykova, M., Yeo, K.: Panorama: Oblivious ram with logarithmic overhead. In: 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS). pp. 871–882. IEEE (2018)
27. Pouliot, D., Wright, C.V.: The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 1341–1352 (2016)
28. Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., Devadas, S.: Constants count: Practical improvements to oblivious RAM. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 415–430 (2015)
29. Sahin, C., Zakhary, V., El Abbadi, A., Lin, H., Tessaro, S.: Taostore: Overcoming asynchronicity in oblivious data storage. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 198–217. IEEE (2016)
30. Stearley, J.: Towards informatic analysis of syslogs. In: 2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935). pp. 309–318. IEEE (2004)
31. Stefanov, E., Dijk, M.v., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. Journal of the ACM (JACM) **65**(4), 1–26 (2018)
32. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious ram. arXiv preprint arXiv:1106.3652 (2011)
33. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS). pp. 299–310. ACM (2013)
34. Vadapalli, A., Henry, R., Goldberg, I.: Duoram: A $Bandwidth-Efficient$ distributed $ORAM$ for 2-and 3-party computation. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3907–3924 (2023)
35. Wu, Z., Li, R.: Obi: a multi-path oblivious ram for forward-and-backward-secure searchable encryption. In: NDSS (2023)
36. Xu, L., Duan, H., Zhou, A., Yuan, X., Wang, C.: Interpreting and mitigating leakage-abuse attacks in searchable symmetric encryption. IEEE Transactions on Information Forensics and Security **16**, 5310–5325 (2021)
37. Xu, L., Zheng, L., Xu, C., Yuan, X., Wang, C.: Leakage-abuse attacks against forward and backward private searchable symmetric encryption. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3003–3017 (2023)

38. Zeng, Z., Veeravalli, B.: Distributed scheduling strategy for divisible loads on arbitrarily configured distributed networks using load balancing via virtual routing. Journal of Parallel and Distributed Computing **66**(11), 1404–1418 (2006)
39. Zeng, Z., Veeravalli, B.: Optimal metadata replications and request balancing strategy on cloud data centers. Journal of Parallel and Distributed Computing **74**(10), 2934–2940 (2014)
40. Zhang, B., Cui, H., Yuan, X., Yu, Z., Guo, B.: $V - ORAM$: A versatile and adaptive $ORAM$ framework with service transformation for dynamic workloads. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 7917–7936 (2025)
41. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of $File - Injection$ attacks on searchable encryption. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 707–720 (2016)
42. Zheng, L., Zhang, Z., Dong, W., Zhang, Y., Wu, Y., Wang, C.: $H2O2RAM$: A $High - Performance$ hierarchical doubly oblivious RAM. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 8501–8520 (2025)
43. Zhu, J., Sun, G., Zhang, X., Zhang, C., Zhang, W., Liang, Y., Wang, T., Chen, Y., Di, J.: Fork path: Batching oram requests to remove redundant memory accesses. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(10), 2279–2292 (2019)
44. Zhu, J., Li, M., Zhang, X., Bu, K., Zhang, M., Song, T.: Hitchhiker: Accelerating oram with dynamic scheduling. IEEE Transactions on Computers **72**(8), 2321–2335 (2023)