Hindawi
Security and Communication Networks
Volume 2022, Article ID 9032828, 12 pages
https://doi.org/10.1155/2022/9032828

Research Article

LPS-ORAM: Perfectly Secure Oblivious RAM with Logarithmic Bandwidth Overhead

Yunping Gong,[1,2] Fei Gao,[1] Wenmin Li,[1] Hua Zhang,[1] Zhengping Jin,[1] and Qiaoyan Wen[1]

[1]*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[2]*State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China*

Correspondence should be addressed to Fei Gao; gaof@bupt.edu.cn

Received 9 May 2022; Accepted 11 July 2022; Published 12 August 2022

Academic Editor: Wenbo Shi

Oblivious Random Access Machine (ORAM) is a cryptographic tool used to obfuscate the access pattern. In this paper, we focus on perfect security of ORAM. A perfectly secure ORAM is an ORAM that can resist against an adversary with unlimited computing power, and the failure probability of ORAM is zero rather than negligible. Since all existing perfectly secure single-server ORAM solutions require at least sublinear worst-case bandwidth overhead, we pose a natural and open question: *can we construct a perfectly secure single-server ORAM with logarithmic worst-case bandwidth overhead?* In this paper, we propose the first tree-based perfectly secure ORAM scheme, named LPS-ORAM. To meet the requirements of perfectly secure ORAM, two techniques are presented. One technique is dynamic remapping associated with a mutable scope, and the other is dynamically balanced eviction. Their combined effect allows the root bucket to never fill up while maintaining its statistical security in tree-based ORAM. In the worst case, our solution achieves logarithmic bandwidth overhead. Therefore, our solution answers the open question in the affirmative. In terms of overhead for temporary storage on the client side, compared with the latest perfectly secure ORAM solution, our solution is reduced from sublinear to logarithmic, and even if the server storage overhead scales lightly, it is still at the same level of quantity as the state of the art. Finally, the evaluation results show that our LPS-ORAM has a significant advantage in terms of bandwidth overhead and overhead for temporary storage on the client side.

1. Introduction

Thanks to the interconnectivity of a large number of mobile smart devices in the Internet of Things, huge amounts of data are being generated. To save money on data storage, consumers choose to store their private data on the cloud server. In order to guarantee confidentiality of the private data, consumers need to encrypt the data before uploading them to the server. But using encryption alone, the data access pattern might still be broken, and the opponent can deduce some sensitive information from this [1–3]. Oblivious Random Access Machine (ORAM) [4–6] was presented decades ago to mitigate this security issue. Nevertheless, these early ORAM solutions are not viewed favorably by most researchers due to the poor efficiency. Since then, a large number of ORAM solutions [7–26] have been put forward to make the efficiency better. Among them, Path ORAM [13] algorithm is very simple and very efficient in logarithmic bandwidth overhead, so it is excellent.

Goldreich and Ostrovsky [6] proposed the first lower bound of $O(\log N)$ bandwidth overhead, where N is the number of real blocks outsourced to the cloud server. The lower bound holds if the client storage size is $O(1)$-block and the ORAM is in a balls-and-bins model with a uniform block size of $O(\log N)$-bit. Boyle and Naor [27] further stated that this lower bound only holds for statistically secure ORAM that is in a "balls-and-bins" manner. Larsen and Nielsen [28] then stated that the lower bound of $O(\log N)$ bandwidth overhead for computationally secure ORAM still holds even it is not in a "balls-and-bins" manner. However, for perfectly secure ORAM, it is not clear whether the lower bound holds.

The theme of a lot of works [13, 27–35] associated with ORAM is to find a lower bound of bandwidth overhead in different settings. Typically, there are two ways to measure the bandwidth overhead. One approach is worst-case overhead, which is the maximum overhead of completing a single request in a long list of requests. The other is amortized-case overhead, which is the average overhead of each request in a long list of requests. The most famous lower bound in the worst case was proposed by Stefanov et al. [13], who presented a tree-based statistically secure ORAM with O ($\log N$) bandwidth overhead when the block size is set to O ($\log^2 N$)-bit. Thus, this only partly matches the lower bound of Goldreich and Ostrovsky [6] because the lower bound only holds when the block size is O ($\log N$)-bit. The most famous lower bound in the amortized case was proposed by Asharov et al. [31], who presented a computationally secure ORAM with O ($\log N$) bandwidth overhead, which completely matches the lower bound of Larsen and Nielsen [28].

The first perfectly secure ORAM was designed by Damgard et al. [36], which has the amortized-case bandwidth overhead of O ($\log^3 N$)-block and the server storage overhead of O ($N * \log N$)-block. This was further improved by Chan et al. [37], who presented a perfectly secure ORAM with a lower server storage overhead of O (N)-block and the same amortized-case bandwidth overhead. Recently, another perfectly secure ORAM, Lookahead ORAM, was proposed by Raskin and Simkin [38], which has the worst-case bandwidth overhead of O ($\sqrt{N}$)-block and the server storage overhead of O (N)-block. This is the first perfectly secure single-server ORAM with sublinear worst-case bandwidth overhead. Since all existing perfectly secure single-server ORAM solutions require at least sublinear worst-case bandwidth overhead, we pose a natural and open question.

Can we construct a perfectly secure single-server ORAM with logarithmic worst-case bandwidth overhead?

This is an important academic question because it facilitates the process of reaching the lower bound for perfectly secure single-server ORAM. Whether this open question can be resolved is a necessary step in the development of ORAM research. The importance of perfectly secure ORAM was elaborated by Chan et al. [37]. In addition to the three points listed, we have added another point. To the best of our knowledge, in the standard model without server computing, the lower bounds of both computationally secure ORAM and statistically secure ORAM are logarithmic, which are Goldreich–Ostrovsky lower bound [6] and Larsen and Nielsen lower bound [28], respectively. However, so far, the lower bound of perfectly secure ORAM has not yet emerged. Therefore, it is significant to keep approaching the lower bound by constructing a perfectly secure ORAM solution with better bandwidth overhead.

1.1. Our Contribution. In this paper, we propose a new perfectly secure ORAM solution, called LPS-ORAM, which is designed to resolve the above open question. The main contributions of our paper are summarized as follows:

(i) Design of LPS-ORAM construction. We propose the detailed design of the first tree-based perfectly secure ORAM construction. The proposed techniques can be applied to implement other perfectly secure tree-based ORAM solutions.

(ii) Simplicity and logarithmic worst-case bandwidth overhead. Our scheme has an extremely simple algorithm that makes it practical to implement, and it gains logarithmic bandwidth overhead in the worst case.

(iii) Small overhead for temporary storage on the client side. Our solution achieves logarithmic overhead for temporary storage on the client side, instead of the previous sublinear. Thus, our solution can be applied to small smart devices with limited client storage in the Internet of Things.

1.2. An Overview of Our Techniques. To the best of our knowledge, if only the requested block is remapped after each access in tree-based ORAM, the root bucket in the ORAM tree will be full sooner or later because if the path corresponding to the remapped leaf label and the eviction path are exactly at two branches of the binary tree, the requested block will have to be evicted into the root bucket. The detailed reason is as follows. It is assumed that the new remapped leaf label of the requested block and the eviction path at that time are exactly at two branches of the binary tree. This is the worst case. Even if the greedy strategy is applied to the eviction process, the requested blocks are continuously allocated to the root bucket, causing the root bucket to accumulate until it is full. To ensure perfect security of ORAM, we need to achieve the goal of allowing the root bucket to never fill up while maintaining its statistical security in tree-based ORAM. As a result, it is not feasible that only requested block is remapped after each access in tree-based ORAM. Thus, dynamic remapping associated with a mutable scope is proposed.

1.2.1. Dynamic Remapping Associated with a Mutable Scope. In tree-based ORAM, to ensure the obliviousness property of ORAM, the new remapped leaf label of the requested block is random and uniform after each access. However, each remaining real block retrieved from the path is remapped to a new leaf label that belongs to the scope from the corresponding leaf label of the eviction path to the original leaf label of the block. This is a dynamic remapping associated with a mutable scope. The scope is mutable because depending on the leaf label of the real block, the real block may be written back into the bucket closer to the leaf bucket, but which bucket in the path the real block is located in is a secret for the honest-but-curious server. For example, assuming that the eviction path at that time is labeled by 3, the original leaf label of a remaining real block is 5, and then the scope at that time is [3, 5]. In our ORAM solution, if the block cannot be directly located to a bucket lower than the original bucket according to the original leaf label, then the new remapped leaf label needs to ensure that it places the bucket in which the block is located closer to the leaf bucket than the original leaf label.

1.2.2. Dynamically Balanced Eviction. In our solution, the goal is allowing the root bucket to never fill up while maintaining its statistical security in tree-based ORAM. Thus, what kind of eviction should be combined with the above dynamic remapping to implement the goal? The dynamically balanced eviction is proposed at this time. During the eviction, the requested block is first arranged to the deepest bucket of the path according to the new remapped leaf label, which is closest to the leaf bucket. Because the new remapped leaf label of the requested block is random and uniform after each access, according to dynamic remapping with a mutable scope, if one path is accessed multiple times, almost all of the real blocks in the path will be squeezed to the buckets near the leaf bucket. This information is harmful because it is inferred easily by the honest-but-curious server. Thus, the dynamically balanced eviction is utilized to avoid the above harmful information. Whether a real block in the path needs to be located in a bucket lower than the original bucket depends on whether the root bucket is empty. If the root bucket is not empty, the above dynamic remapping needs to be executed. Otherwise, it cannot be executed. Thus, the proposed eviction is dynamically balanced.

1.3. Other Related Work. A great deal of work has contributed to implementing perfectly secure ORAM. In the rest of this section, we provide only a high-level overview of solutions that are directly relevant to our work.

In order to reach the lower bound of $O(\log N)$, a large number of solutions have been proposed in the client-server environment. Mayberry et al. [39] proposed a solution with server-side computations, called Path-PIR, in order to obtain a actually very small, but still polylogarithmic bandwidth overhead. Apon et al. [40] formally defined the primitive for verifiable oblivious storage by allowing server-side computations to be generated from the ORAM primitive and by providing a solution with constant bandwidth overhead, but it is based on fully homomorphic encryption (FHE), so it shows that $O(\log N)$ lower bound has been broken in their setting with FHE. Another solution, called Onion ORAM [41], is proposed that also breaks the $O(\log N)$ lower bound, but it relies on additively homomorphic encryption (AHE). However, in this work, we will focus on the client-server setting without server-side computations.

Demertzis et al. [42] proposed a computationally secure ORAM solution with worst-case bandwidth overhead of $O(N^{1/3})$ and perfect correctness. Perfect correctness means that the ORAM solution fails with the probability of 0. Subsequently, several works cite this and claim that their solution is perfectly secure. However, according to their definition of perfectly secure ORAM, Raskin and Simkin [38] stated that this is not correct and this claim is not made by the authors of that paper either.

1.4. Organization. The rest of this paper is organized as follows. Section 2 introduces the background knowledge including the definition of perfectly secure ORAM and an overview of Path ORAM. Section 3 provides the details of our LPS-ORAM solution. Section 4 gives the performance analysis in terms of bandwidth overhead, storage overhead, and further optimization. A detailed evaluation is introduced in Section 5. Finally, the conclusion is provided in Section 6.

2. Preliminaries

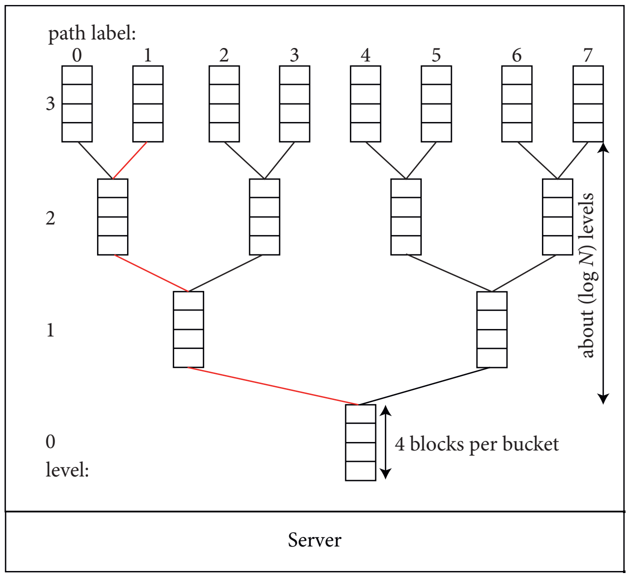
2.1. Security Model. In the security model of ORAM, it is assumed that there is an honest-but-curious server and a trusted client. It requires that for any two requests sequences with the same length, the corresponding access pattern should be indistinguishable. Note that all blocks are encrypted by the client before they are uploaded to the server. The following security definition of perfectly secure ORAM is taken from Raskin and Simkin [38].

Definition 1. (security definition of perfectly secure ORAM). Let $\overrightarrow{U} = (Y_1, Y_2, Y_3, \ldots)$ indicate a request sequence of ORAM. In $\overrightarrow{U}$, Y_i is an access of Read (ID_i) or Write (ID_i, $\mathrm{Data}_i{}^*$), where ID_i means the unique block identifier and $\mathrm{Data}_i{}^*$ refer to the new content of block ID_i to be written. It is noted that each real block has a unique identifier. Let DAP ($\overrightarrow{U}$) represent the data access pattern when $\overrightarrow{U}$ is the input of the ORAM algorithm. In reality, the data access pattern is viewed as a distribution. The ORAM solution is statistically/computationally secure for the honest-but-curious server, if and only if DAP ($\overrightarrow{U}$) and DAP ($\overrightarrow{V}$) are statistically/computationally indistinguishable for any two ORAM request sequences $\overrightarrow{U}$ and $\overrightarrow{V}$ with the same length. The ORAM solution is perfectly correct if and only if it returns on input $\overrightarrow{U}$ that is consistent with $\overrightarrow{U}$ with probability 1. We call an ORAM perfectly secure if and only if the ORAM solution can resist against an adversary with unlimited computing power and is perfectly correct at the same time.

According to the above definition, the ORAM is perfectly secure if an ORAM is statistically secure and has a failure probability of 0 at the same time.

2.2. An Overview of Path ORAM. We provide a simple overview of Path ORAM (see [13] for more details). As described in Figure 1, the Path ORAM solution consists of two parts, one is the server storage, and the other is the client storage. The server storage is a complete binary tree with about log N-level. The red line is a target path that the requested block is stored, which is from the remapped leaf label of the position map (PosMap) on the client.

In the complete binary tree, each node is a bucket that can accommodate at most Z-block where Z is a constant. Z-block contains some real blocks, and the rest of the space is populated with virtual blocks. The difference between a real block and a virtual block is that the content of the virtual block is a random string, while the content of the real block consists of real data. Each path in the complete binary tree is



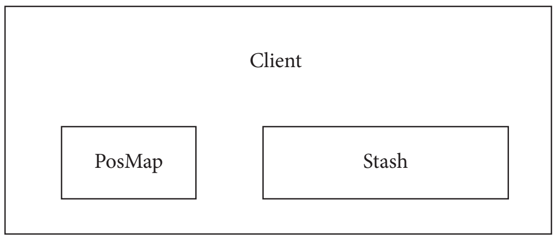


Figure 1: The structure of Path ORAM solution [13].

a set of buckets from the root bucket to a leaf bucket. After each access, every requested block is remapped to a random and uniform leaf label, which means that the requested block either resides somewhere on the path numbered by the leaf label or in stash on the client side. In Path ORAM, to execute an ORAM request, the PosMap is queried first by the client, which is a list table on the client side that tracks the path to which each real block is currently remapped, and then about ($Z * \log N$) blocks on that path are retrieved to the local stash. Subsequently, the requested block is remapped to a new random and uniform leaf label and the PosMap is updated accordingly. Finally, the eviction procedure is executed, the same path is populated with some real blocks, and the rest of the space is populated with virtual blocks. The various symbols and their meanings are listed in Table 1.

The bandwidth overhead of Path ORAM is about $2Z * \log N$ because a path is fetched and then it is written back into the complete binary tree for each ORAM request. To make Path ORAM fail with a negligible probability in N, the value of Z must be at least 4 in reality or 5 in theory.

3. LPS-ORAM Solution

In this section, we present an extremely simple tree-based perfectly secure ORAM protocol. As far as we know, Path ORAM has an extremely simple algorithm and efficient efficiency of $O(\log N)$-block bandwidth overhead when the block size is set to $O(\log^2 N)$-bit. In our design, our perfectly secure tree-based ORAM, called LPS-ORAM, will inherit the benefits of Path ORAM. In addition, our works focus on perfect security of ORAM, that is, we are committed to achieving the goal of allowing the root bucket to never fill up while maintaining its statistical security in tree-based ORAM. The various symbols used in this solution and their meanings are also listed in Table 1.

Table 1: Symbols and meanings.

Symbol	Meaning
N	The number of real blocks in total
L	The height of the full binary tree
Z	The bucket size in blocks
B	The block size in bits
p	Path p refers to the set from the root bucket to leaf bucket labeled p
$P(j, i)$	The bucket at level i along the path labeled j

3.1. Storage Structure. In our LPS-ORAM, there are N real blocks, which are outsourced to the server storage. Each block's modality is (*identifier*, p; *data*). It represents that the block numbered as *identifier* is either on the path numbered by leaf label p or in the local stash. For each real block, it has a unique identifier, *identifier*, and the content of the block labeled as *identifier* contains *data*. For each virtual block, both p and *data* are populated with random strings. In order to obfuscate all the blocks with each other, they are set to a constant size no matter whether the block is a real block or a virtual block. To differentiate decrypted blocks retrieved from the server storage, all virtual blocks have same block identifiers. In addition, the purpose of adding virtual blocks to the server storage is to confuse all blocks so that the server cannot differentiate between any encrypted block being a real block or a virtual block. Note that all blocks are encrypted by the client before they are uploaded to the server. That is, all blocks in the server storage are in the state of encryption. Therefore, adding virtual blocks to the server storage is part of the security effort to hide the data access pattern.

3.1.1. Server Storage. On the server side, there is a complete binary tree. In it, there are $L+1$ levels in total. They are marked as 0, 1, 2, . . ., and L, respectively. Theoretically, the height of the complete binary tree is set to $L = \lceil \log N \rceil + 1$. For the sake of description, we let $L = \log N$, resulting in a full binary tree with N leaves and $N-1$ non-leaves. In the complete binary tree, the root node is at layer 0 and all the leaf nodes are at layer L.

Each node of the complete binary tree is one bucket in our LPS-ORAM solution. Z blocks at most in each bucket. As a result, each bucket is Z-block in size. In our scheme, Z is set to a constant. Each bucket can accommodate some real blocks, and dummy blocks populate the rest of the space.

The complete binary tree has about $2N$ nodes, so there are about $2Z * N$ blocks on the server side. That is, the server storage size is $O(N)$ blocks.

```
Algorithm: Access (a, op, data*)

// The requested block a is retrieved and operated.

// The algorithm consists of the Retrieval algorithm and the Eviction algorithm.

1.  j ←PosMap(a)                    // lookup the PosMap locally

2.  fori= 0, 1, 2, …, H do //Fetch all buckets of the target path
3.  Stash←Stash+P (j, i)      // get the requested block a
4.  end for

5.  data← Fetch block a from Stash
6.  ifop = 'write'
7.  thendata←data*
8.  end if

9.  PosMap(a)←Uniform_And_Random (0, N]
    //assuming that all identifiers of dummy blocks are set to 0.
10. get the leaf label of each fetched real block
11. get the mutable scope of each fetched real block
12. If the real block can be directly written back into a lower bucket than the original bucket
13.   Then the block need not to be remapped
14. Else
15.   If the root bucket is not empty
16. Theneach fetched real block is remapped to a new leaf label from the above scope, until the real block can be
    written back into a lower bucket than the original bucket.
17.    end if
18. end if

19. fori= L, L- 1, …, 0 do   // scheduled from the corresponding leaf bucket to the root bucket.
20.     Each real block is written back into the bucket as close to the leaf bucket as possible.
21. Write back P(j) into the binary tree.
22. end for

23. returndata
```

Figure 2: The algorithm of our LPS-ORAM solution.

3.1.2. Client Storage. There are two structures on the client side, PosMap and stash.

In our scheme, there are N real blocks, and each real block is remapped to a leaf label, so there are N leaf labels. All of the above N leaf labels are stored in the PosMap. The complete binary tree has N leaves, and each leaf is numbered by a leaf label, which results in each leaf label having a size of $\log N$ bits. Thus, the size of the PosMap is $(N * \log N)$ bits. In the client-server environment, the client can entirely store the PosMap, rather than the server storing the PosMap recursively because storing the PosMap on the client is virtually negligible when the block size is not set to very small size of $O(\log N)$ bits. Moreover, if the server stores the PosMap recursively, both the average time latency and the number of interaction rounds increase significantly.

The client has a stash to store temporary blocks retrieved from the server storage. Why is it temporary storage? Because all blocks on the target path are retrieved to the local stash, and then all the blocks are written back into the path of the complete binary tree. In the previous tree-based ORAM solutions, there might have been some stranded real blocks on the stash because the root bucket might have been full. In our perfectly secure tree-based ORAM solution, the stash size is exactly the size of retrieved path because there are no stranded real blocks on the stash.

3.2. Detailed Execution. In this section, the detailed execution procedures are described as follows. There are two algorithms to implement our LPS-ORAM protocol, which are retrieval algorithm and eviction algorithm, respectively. A detailed description of the whole LPS-ORAM algorithm is shown in Figure 2.

3.2.1. Retrieval. The retrieval algorithm is to fetch all blocks on the target path corresponding to the leaf label of the requested block. All the fetched blocks are stored in the stash locally. After the retrieval, all the fetched blocks are decrypted and then the dummy blocks are discarded. That is, only the real blocks are stored in the stash on the client. Subsequently, the requested block is assigned to a new leaf label from random and uniform remapping. In order to implement a failure probability of 0, that is, to allow the root bucket to never fill up while maintaining its statistical security in tree-based ORAM, all remaining fetched real blocks need to be remapped if necessary. However, instead of applying a random and uniform remapping to them, a new remapping is applied to them. Since the distribution of real blocks on the path in which buckets is dynamic and the server cannot know the distribution, we can take the step of infiltrating the real block down the position of a bucket to free up space of the upper bucket. Thus, the root bucket will not be full even if the remapping of the requested block is in the worst case, where both the path corresponding to the leaf label of the requested block and the eviction path at that time are two branches of the binary tree. The question is what kind of remapping would allow fetched real blocks to move down one bucket? At this point, the dynamic remapping associated with a mutable scope is proposed.

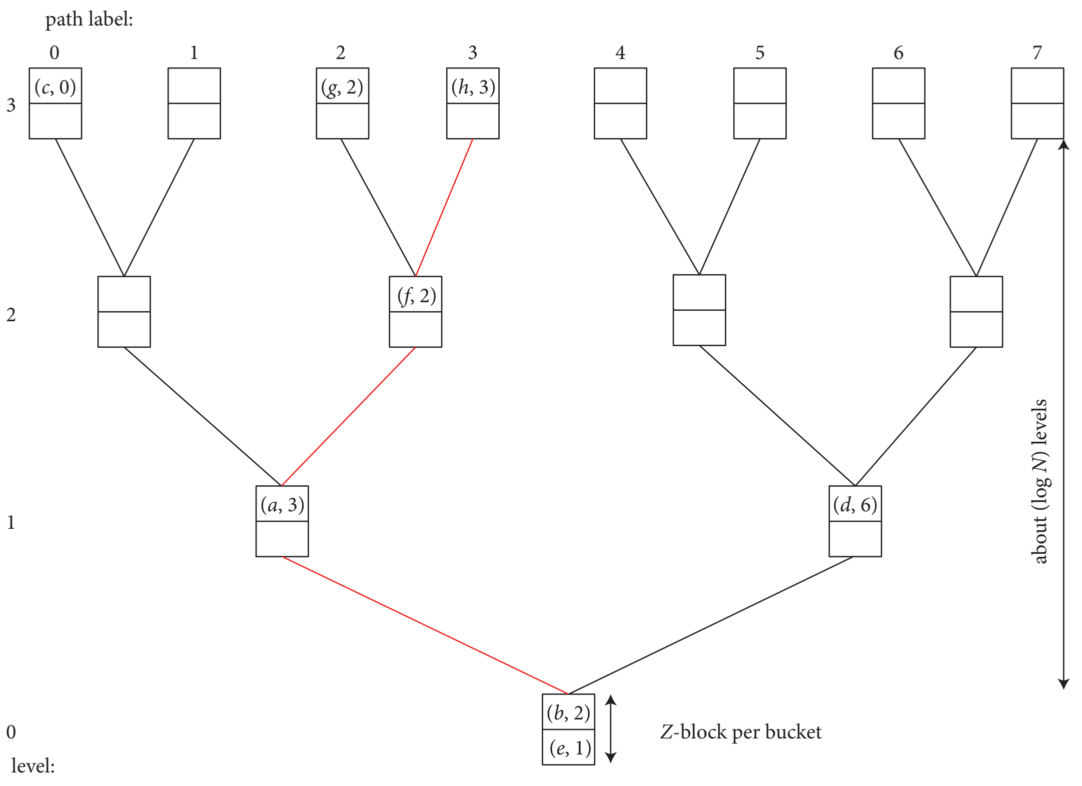
Now we illustrate our proposed dynamic remapping with a mutable scope, as shown in Figure 3. The target path is marked by read line and taken from the PosMap on the client, and all real blocks in the target path are $(a, 3)$, $(b, 2)$, $(e, 1)$, $(f, 2)$, $(h, 3)$. It is noted that data of each real block is ignored to descript simply. For example, block (a, 3, data) is written as (a, 3). Among the fetched real blocks, block $(a, 3)$ is the requested block at that time. Since the root bucket is not empty and block (f, 2) cannot be located in a bucket lower than the original bucket, block $(f, 2)$ has to be remapped from a mutable scope [2, 3], which is the set from the original leaf label to the leaf label corresponding to the eviction path. In Figure 3, it is marked in yellow. Also, the rest of the fetched real blocks $(b, 2)$, $(e, 1)$, $(h, 3)$ need not be remapped because they can be directly written back into a lower bucket than the original bucket. The requested block $(a, 3)$ is remapped to a new random and uniform leaf label because of the obliviousness property of ORAM.

As shown in Figure 2, step 1 is the lookup procedure to get the leaf label of the target path, which is the leaf label of the requested block. From step 2 to step 4 is the retrieval procedure to get all buckets of the target path in the binary tree. Then, from step 5 to step 8 is the operation procedure of the requested block during each access. If the operation is "*read*," *data* of the requested block are directly returned to the client, as described in step 23. If the operation is "*write*," *data* are updated by $data*$. From step 9 to step 18 is the remapping procedure to get a new leaf label for each fetched real block. The procedure is divided into two cases. One case is for the requested block, and the other case is for all other fetched real blocks from the target path. Due to the obliviousness property of ORAM, after each access, the requested block needs to be remapped to a new random and uniform leaf label. Nevertheless, all other fetched real blocks can avoid the random and uniform remapping because they are dynamically distributed on each path and this distribution is secret for the server. Thus, it is secure to adjust the distribution of them to achieve some certain goal. If the root bucket is not empty, each real block in the path will be written back into a lower bucket than the original bucket through adjusting the corresponding leaf label. Note that a lower bucket is closer to the corresponding leaf bucket in the binary tree. If some real block of them can be directly written back into a lower bucket than the original bucket, the block need not be remapped. Else, the block is remapped to a new leaf label from the above scope, until the real block can be written back into a lower bucket than the original bucket, this process takes almost no time.

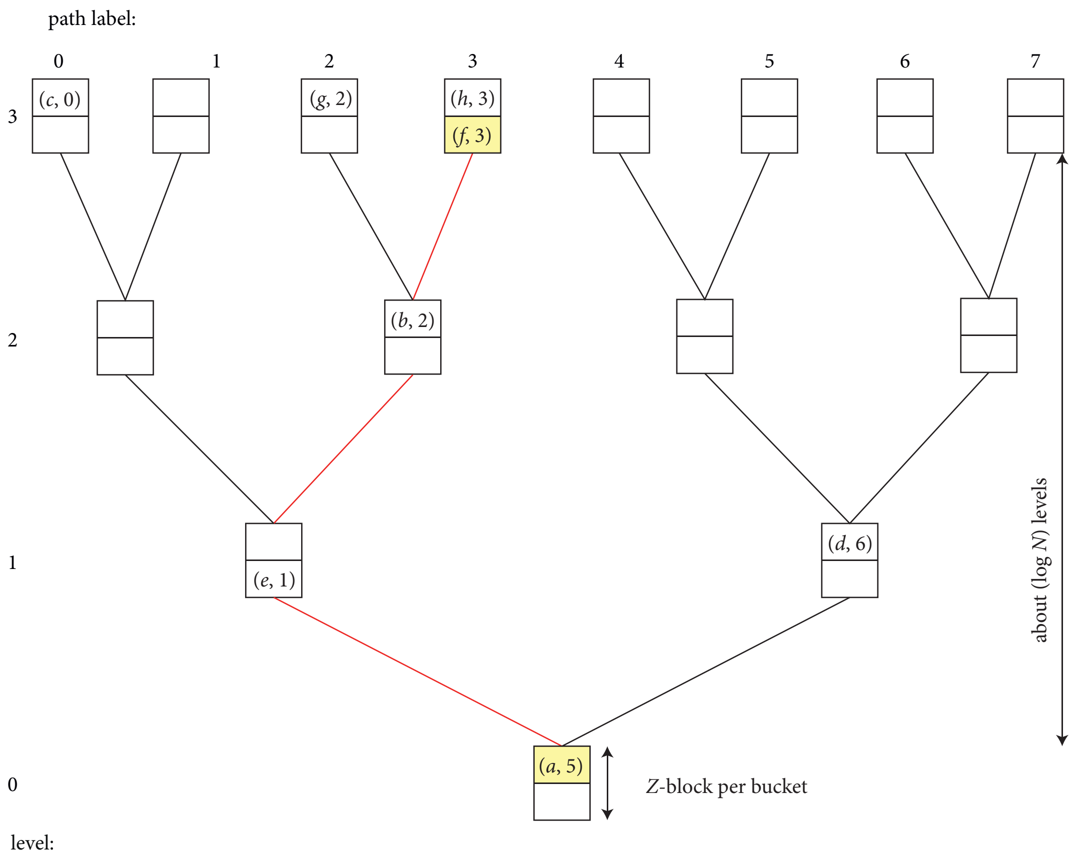
3.2.2. Eviction. In this section, the dynamically balanced eviction algorithm is proposed. The proposed eviction algorithm not only follows the greed strategy but also further makes use of the space of the eviction path in the binary tree. The greed strategy in the eviction algorithm is that as many fetched blocks as possible are written back from the stash locally to the eviction path in the binary tree. The above dynamic remapping associated with a mutable scope can make each real block locate into a lower bucket. Thus, our dynamically balanced eviction algorithm can be combined with the above dynamic remapping associated with a mutable scope to make better use of the space of the eviction path in the binary tree than a single greed strategy.

During the eviction algorithm, the requested block is first arranged to the deepest bucket of the path according to the new remapped leaf label, which is closest to the leaf bucket. Because the new remapped leaf label of the requested block is random and uniform after each access, according to dynamic remapping with a mutable scope, if one path is accessed multiple times, almost all of the real blocks in the path will be squeezed to the buckets near the leaf bucket. This information is harmful because it is inferred easily by the honest-but-curious server. Thus, the dynamically balanced eviction is utilized to remove the above harmful information. Whether a real block in the path needs to be located into a bucket lower than the original bucket, it depends on whether the root bucket is empty. If the root bucket is not empty, the above dynamic remapping needs to be executed. Otherwise, it cannot be executed. Thus, the proposed eviction is dynamically balanced. In our eviction algorithm, for each access, the goal is that the requested block can be written back into the path in the binary tree, rather than being stranded in the stash locally. This goal is the focus of our scheme in the setting of tree-based ORAM. In tree-based ORAM, the root bucket may be full because only the requested block needs to be remapped to a random and uniform leaf label, while other fetched real blocks do not need to be remapped. If the requested block for each access is in the worst case, with a small but non-negligible probability, the root bucket will accumulate until it is full as the number of different requested blocks increases. However, our eviction algorithm can avoid this case.

As shown in Figure 2, from step 19 to step 22 is the procedure of the eviction algorithm to write back all fetched real blocks containing the requested block into the eviction path in the binary tree. During this procedure, the path is scheduled from the corresponding leaf bucket to the root bucket. Then, each fetched real block is written back into some lower bucket than the original bucket if the root bucket is not empty. Finally, the eviction path is written back into the binary tree on the server storage.



(a)



(b)

Figure 3: The proposed dynamic remapping. (a) The state of the binary tree before the proposed dynamic remapping. The red line is the target path. The block of bold lines is the requested block labeled a, namely, block $(a, 3)$, where *data* is ignored to descript simply. (b) The state of the binary tree after eviction. The fetched real blocks that are not marked yellow need not to be remapped, such as blocks $(b, 2)$, $(e, 1)$, $(h, 3)$, while the fetched real blocks marked yellow except the requested block need to follow the dynamic remapping with a mutable scope, for example, block $(f, 2)$ is modified to $(f, 3)$. The requested block $(a, 3)$ is remapped to a new random and uniform leaf label 5 because of the obliviousness property of ORAM.

3.3. Security Analysis. In this section, we will analyze the perfect correctness and perfect security of our LPS-ORAM. The perfect correctness of ORAM means that the ORAM scheme fails with the probability of 0, rather than a negligible probability. The perfect security of ORAM means that the ORAM scheme can resist against an adversary with unlimited computing power, and simultaneously the ORAM scheme has perfect correctness.

3.3.1. Perfect Correctness

Claim 1. Our LPS-ORAM scheme has perfect correctness.

Proof. If the block can be directly written back into a lower bucket than the original bucket, the fetched real block need not be remapped. Else, each fetched real block is remapped to a new leaf label from a mutable scope, until the real block can be written back into a lower bucket than the original bucket. In a word, each fetched real block from the target path needs to be located in a lower bucket than the original bucket. However, the requested block needs to follow the random and uniform remapping because of the oblivious property of ORAM. When the requested block is in the worst case, namely, both the path corresponding to the new leaf label of the requested block and the eviction path are two branches of the binary tree, the requested block will have to be written back into the root bucket. However, since each of all other real blocks can be written back into a lower bucket than the original bucket, the root bucket is filled with at most one real block at any epoch. As a result, as long as the size of the root bucket is larger than one block, the root bucket cannot be full. That is, our LPS-ORAM scheme can fail with the probability of 0 as long as the bucket size is longer than 1. Therefore, our LPS-ORAM scheme has perfect correctness. □

3.3.2. Perfect Security

Claim 2. Our LPS-ORAM scheme is statistically secure for the honest-but-curious server.

Proof. In our LPS-ORAM, each path fetched is random and uniform for the honest-but-curious server. That is, all blocks of each bucket fetched from the binary tree are random and uniform. As a result, for any two kinds of access, the two paths retrieved are statistically indistinguishable for the server. Therefore, our LPS-ORAM scheme is statistically secure for the honest-but-curious server. □

Theorem 1. *Our LPS-ORAM scheme is perfectly secure for the honest-but-curious server.*

Proof. According to Claim 2, our LPS-ORAM is statistically secure for the honest-but-curious server. As a result, our scheme can resist against an adversary with unlimited computing power. In addition, according to Claim 1, our LPS-ORAM scheme has perfect correctness. Therefore, according to the security definition from Definition 1, our LPS-ORAM scheme is perfectly secure for the honest-but-curious server. □

4. Performance Analysis

In this section, we will analyze the asymptotic performance, which is mainly in bandwidth overhead and storage overhead. We proposed the measures of further optimization. Our LPS-ORAM solution will be compared with all the previous perfectly secure single-server ORAM solutions, which are listed in Table 2.

4.1. Bandwidth Overhead. In our solution, for each access, only one path is fetched and then is written back into the binary tree. Thus, to fetch a requested block, the number of blocks transferred between the client and the server is O (log N) blocks. As a result, the bandwidth overhead of our solution is O (log N)-block.

4.2. Storage Overhead. In our solution, the bucket size Z is a constant and the binary tree on the server storage has O (N) buckets. As a result, the binary tree has O (N) blocks. That is, the server storage overhead of our solution is O (N)-block. The client storage consists of PosMap and stash. PosMap is practically negligible in the setting of client-server, as mentioned in S^{3}ORAM [26]. Thus, the stash size is the client storage overhead, which is one path size. Therefore, the client storage overhead of our solution is O (log N)-block.

4.3. Further Optimization. In our LPS-ORAM solution, if the bucket size Z is set to 1, the asymptotic performance of our solution can be further optimized. In this case, the bandwidth overhead is $(L+1)$-block, the server storage overhead is $(2^{L+1}-1)$-block, and the client storage overhead is one path size of $(L+1)$-block. So, these overheads are determined by the value of L. In theory, the value of L is set to $\lceil \log N \rceil + 1$. So, the number of buckets is about $4N$. There is enough space to percolate down for the real blocks to release the root bucket. However, to release the root bucket more likely, a larger number of buckets or a larger Z is needed. Thus, the value of L and Z is in a dynamic equilibrium to achieve a trade-off.

5. Evaluation

To give the actual performance of our LPS-ORAM solution, we implemented a prototype with a client-side position map and evaluated it based on bandwidth overhead, temporary storage overhead, and server storage overhead. Our solution will be compared with all the previous perfectly secure single-server ORAM solutions. So far, there are three such solutions. They are proposed by Damgard et al. [36], Chan et al. [37], and Raskin and Simkin [38], respectively. In addition, Path ORAM solution [13] is also compared with ours because it is a tree-based ORAM with efficient efficiency of logarithmic bandwidth overhead.

In comparison, for different values of N, we measure the bandwidth overhead, namely, the total amount of data

Table 2: Asymptotic performance comparison of all the perfectly secure single-server ORAM schemes.

Perfectly secure ORAM scheme	Structure	Amortized-case bandwidth	Worst-case bandwidth	Client storage	Server storage
Damgard et al. [36]	Layer	$O\ (\log^3 N)$	$O\ (N * \log N)$	$O\ (1)$	$O\ (N * \log N)$
Chan et al. [37]	Layer	$O\ (\log^3 N)$	$O\ (N * \log N)$	$O\ (1)$	$O\ (N)$
Raskin et al. [38]	Matrix	$O\ (\sqrt{N})$	$O\ (\sqrt{N})$	$O\ (\sqrt{N})$	$O\ (N)$
Ours	Tree	$O\ (\log N)$	$O\ (\log N)$	$O\ (\log N)$	$O\ (N)$

Note. All asymptotic overheads are represented in blocks.

transferred per access between the client and the server. In addition, both the temporary storage size on the client and the total storage size on the server are measured in our experiments. In the respective works, the values in all the compared ORAM schemes with our LPS-ORAM are calculated based on the concrete formulas and constants that are reported.

We make the following assumptions, as mentioned in the evaluation of Lookahead ORAM solution [38]. There is an additional encryption/MAC overhead of about 40 bytes in each encrypted block because the random encryption is applied to all blocks of all ORAM schemes. Within each stash slot, there is an additional state header of about 20 bytes that contains location information and the state. Also, 4-byte words are used to indicate the ORAM request types. During initialization procedure, the storage is populated with random strings and they are directly uploaded to the server. In all solutions, the block size is fixed to 1024 bytes.

We first analyze the concrete value of bandwidth overhead, temporary storage overhead, and server storage overhead in our LPS-ORAM solution. In it, $L = \lceil \log N \rceil + 1$ in theory and $Z = 1$. Thus, when N real blocks of each size B-bit are encrypted, the total server storage overhead of the server is $Z \times (2^{L+1} - 1) \times (B + 40)$-bit $= (4N - 1) \times (B + 40)$-bit. The corresponding position map is $(N \times \log N)$-bit. The stash size is $(B + 40) \times (2 + \log N)$-bit, which is the temporary storage overhead. For each access, $(B + 40) \times (2 + \log N)$ bits need to be downloaded and then $(B + 40) \times (2 + \log N)$ bits need to be uploaded, and thus the bandwidth overhead is $2 \times (B + 40) \times (2 + \log N)$ bits.

Finally, for the sake of description in the following figures, the solutions proposed by Damgard et al. [36] and Chan et al. [37] are called ORAM_1, ORAM_2, and the solution proposed by Raskin and Simkin [38] is called Lookahead ORAM.

5.1. Bandwidth Overhead.

The bandwidth overhead refers to the number of blocks transferred between the client and the server to obtain a requested block. The bandwidth overheads of the above compared solutions are listed in the following.

The ORAM_1 and ORAM_2 solutions are based on a hierarchical structure, which have no position map on the client. Their concrete bandwidth overheads are $(\log^2 N) * (1 + \log N)/2$ blocks, which are self-reported. As a result, the value is $(B + 40) \times (\log^2 N) \times (1 + \log N)/2$ bits. The Lookahead ORAM is based on a matrix structure, which also has a position map on the client. The position map is also $O\ (N \times \log N)$-bit. However, the recursion technique is not considered to be applied to the position map. Thus, its concrete bandwidth overhead is $\{40 + (B + 40) \times (\sqrt{N} + 1)\} + \{80 + (B + 40) \times (\sqrt{N} + 1)\} = 120 + 2 \times (B + 40) \times (\sqrt{N} + 1)$ bits, which is self-reported. In addition, the bandwidth overhead of Path ORAM is also shown in Figure 4. In Path ORAM, the bandwidth overhead $= (B + 40) \times 2 \quad Z \times \log \quad N = 10 \times (B + 40) \times \log N$ bits, as shown in the evaluation of Lookahead ORAM.

Finally, the results are shown in Figure 4. As expected, our LPS-ORAM has the smallest bandwidth overhead of all the compared solutions.

5.2. Temporary Storage Overhead.

The temporary storage overhead on the client side refers to the number of blocks stored on the client, which is temporary because after each access, all fetched blocks stored in the stash locally are written back into the server storage. The temporary storage overheads of the above compared solutions are listed in the following.

The ORAM_1 and ORAM_2 solutions are based on a hierarchical structure. The temporary storage overheads contain one block, which is self-reported. As a result, the value is $(B + 40)$ bits. The Lookahead ORAM is based on a matrix structure, which also has a position map on the client. Its concrete temporary storage overhead on the client is $80 + (B + 40) \times (\sqrt{N} + 1)$ bits, which is self-reported. Additionally, in Path ORAM, the temporary storage overhead on the client is about $10\,N \times (B + 40)$ bits, which is self-reported.

We observed that the temporary storage overhead is about half of the bandwidth overhead in Lookahead ORAM, Path ORAM, and our solution, while the temporary storage overheads in the ORAM_1 and ORAM_2 solutions are only considered to be a small constant. Thus, the figure of results for Lookahead ORAM, Path ORAM, and our solution is similar to that of Figure 4. As expected, our LPS-ORAM has the smallest temporary storage overhead among the above three solutions.

5.3. Server Storage Overhead.

The storage overhead on the server side refers to the number of blocks stored on the server, which not only contains real blocks but also dummy blocks. The storage overheads of the above compared solutions are listed in the following.

The ORAM_1 solution is based on a hierarchical structure. Its concrete storage overhead of the server is $(2N - 1) \times \log N$ blocks, which is self-reported. As a result, the value is $(B + 40) \times (N - 1) \times \log N$ bits. The ORAM_2 solution is based on the ORAM_1 solution. Its concrete storage overhead of the server is reduced to $2N$ blocks, which is self-reported. As a result, the value is $(B + 40) \times 2N$ bits. The Lookahead ORAM

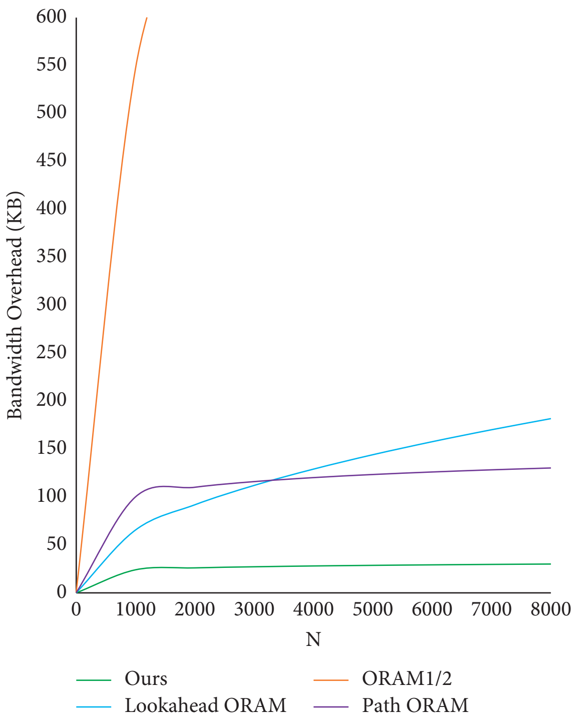


FIGURE 4: Comparison of the bandwidth overheads of all the perfectly secure single-server ORAM solutions. The X-axis shows different values of N. The Y-axis shows the total amount of data transferred per access in KB between the client and the server.

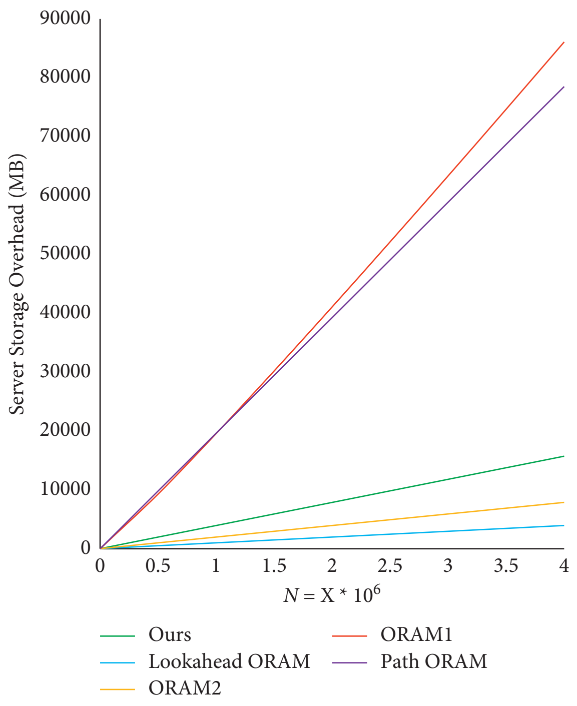


FIGURE 5: Comparison of the server storage overheads of all the perfectly secure single-server ORAM solutions. In the X-axis, $N = X * 10^6$. The Y-axis shows the total required storage on the server side in MB.

is based on a matrix structure, which also has a position map on the client. The position map is also $O\ (N * \log N)$-bit. However, the recursion technique is also not considered to be applied to the position map. Thus, its concrete storage overhead of the server is $N \times (B + 40)$ bits, which is self-reported. In addition, the serve storage overhead of Path

ORAM is also shown in Figure 5. In Path ORAM, the server storage overhead is about $20\,N \times (B + 40)$ bits, as shown in the evaluation of Lookahead ORAM.

Finally, the results are shown in Figure 5. As expected, our LPS-ORAM is slightly larger than Lookahead ORAM in terms of server storage overhead.

6. Conclusion

In this paper, we focus on perfect security of ORAM. Since all existing perfectly secure single-server ORAM solutions require at least sublinear worst-case bandwidth overhead, a natural and open question is posed: *can we construct a perfectly secure single-server ORAM with logarithmic worst-case bandwidth overhead*? To affirmatively answer the question, we propose the first tree-based perfectly secure ORAM scheme with logarithmic worst-case bandwidth overhead, called LPS-ORAM. To meet the requirements of perfectly secure ORAM, two techniques are used. One technique is dynamic remapping associated with a mutable scope, and the other is dynamically balanced eviction. Their combined effect allows the root bucket to never fill up while maintaining its statistical security in tree-based ORAM. In terms of overhead for temporary storage on the client side, compared with the latest perfectly secure ORAM solution, our solution is reduced from sublinear to logarithmic, even if the server storage overhead scales lightly, it is still at the same level of quantity as the state of the art. Finally, the evaluation results show that our LPS-ORAM has a significant advantage in terms of bandwidth overhead and overhead for temporary storage on the client side.

Data Availability

The data used to support the findings of this study are available from the authors upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This study was supported by the National Key R&D Program of China under grant no. 2020YFB1005900 and National Natural Science Foundation of China under grant no. 62072051.

References

[1] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation," in *Proceedings of the 2012 Network and Distributed System Security Symposium*, pp. 1–15, San Diego, CA, USA, February 2012.

[2] J. L. Dautrich and C. V. Ravishankar, "Compromising privacy in precise query protocols," in *Proceedings of the 16th International Conference on Extending Database Technology - EDBT '13*, pp. 155–166, Genoa Italy, March 2013.

[3] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pp. 668–679, Denver, CO, USA, October 2015.

[4] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*, pp. 182–194, New York, NY, USA, 1987.

[5] R. Ostrovsky, "Efficient computation on oblivious RAMs (extended abstract)," in *Proceedings of the STOC '90 Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pp. 514–523, Baltimore, MD, USA, May 1990.

[6] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[7] S. Y. Chiou and Y. X. He, "Generalized proxy oblivious signature and its mobile application," *Security and Communication Networks*, vol. 2021, Article ID 5531505, 16 pages, 2021.

[8] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Advances in Cryptology – CRYPTO 2010*, vol. 6223, pp. 502–519, Springer, Berlin, Heidelberg, 2010.

[9] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with O((log N)^3) worst-case cost," in *Advances in Cryptology – ASIACRYPT 2011*, vol. 7073, pp. 197–214, Springer, Berlin, Heidelberg, 2011.

[10] Z. Li, C. Xiang, and C. Wang, "Oblivious transfer via lossy encryption from lattice-based cryptography," *Wireless Communications and Mobile Computing*, vol. 2018, Article ID 5973285, 11 pages, 2018.

[11] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *Proceedings of the 2012 Network and Distributed System Security Symposium*, pp. 1–40, San Diego, CA, USA, February 2012.

[12] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (In)security of hash-based oblivious RAM and a new balancing scheme," in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 143–156, San Francisco, CA, USA, January 2012.

[13] E. Stefanov, M. Van Dijk, E. Shi et al., "Path ORAM: an extremely simple oblivious RAM protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pp. 299–310, Berlin, Germany, November 2013.

[14] S. Zhao, X. Song, H. Jiang, M. Ma, Z. Zheng, and Q. Xu, "An efficient outsourced oblivious transfer extension protocol and its applications," *Security and Communication Networks*, vol. 2020, Article ID 8847487, 12 pages, 2020.

[15] S. Gordon, X. Huang, A. Miyaji, C. Su, K. Sumongkayothin, and K. Wipusitwarakun, "Recursive matrix oblivious RAM: an ORAM construction for constrained storage devices," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 12, pp. 3024–3038, 2017.

[16] H. Ding, H. Jiang, and Q. Xu, "Postquantum cut-and-choose oblivious transfer protocol based on LWE," *Security and Communication Networks*, vol. 2021, Article ID 9974604, 15 pages, 2021.

[17] X. Zhang, G. Sun, C. Zhang et al., "Fork path: improving efficiency of ORAM by removing redundant memory accesses," in *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, pp. 102–114, Waikiki, Hl, USA, December 2015.

[18] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: a dissection and experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, 2016.

[19] B. Li, Y. Huang, Z. Liu, J. Li, Z. Tian, and S.-M. Yiu, "HybridORAM: practical oblivious cloud storage with constant bandwidth," *Information Sciences*, vol. 479, pp. 651–663, 2019.

[20] Z. Liu, Y. Huang, J. Li, X. Cheng, and C. Shen, "DivORAM: towards a practical oblivious RAM with variable block size," *Information Sciences*, vol. 447, pp. 1–11, 2018.

[21] J. Sancho, J. García, and A. Alesanco, "Oblivious inspection: on the confrontation between system security and data privacy at domain boundaries," *Security and Communication Networks*, vol. 2020, Article ID 8856379, 9 pages, 2020.

[22] X. Yu, S. K. Haider, L. Ren et al., "PrORAM: dynamic prefetcher for oblivious RAM," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pp. 616–628, Portland, OR, USA, June 2015.

[23] H. Yang, J. Shen, J. Lu, T. Zhou, X. Xia, and S. Ji, "A privacy-preserving data transmission scheme based on oblivious transfer and blockchain technology in the smart healthcare," *Security and Communication Networks*, vol. 2021, Article ID 5781354, 12 pages, 2021.

[24] Y. Ishai, E. Kushilevitz, R. Ostrovsky, M. Prabhakaran, and A. Sahai, "Efficient non-interactive secure computation," in *Advances in Cryptology – EUROCRYPT 2011*, vol. 6632, pp. 406–425, Springer, Berlin, Heidelberg, 2011.

[25] L. Ren, C. Fletcher, A. Kwon et al., "Constants count: practical improvements to oblivious RAM," in *Proceedings of the 24th USENIX Security Symposium*, pp. 415–430, Washington, D.C, USA, August 2015.

[26] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. 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 '17*, pp. 491–505, Dallas, TX, USA, November 2017.

[27] E. Boyle and M. Naor, "Is there an oblivious RAM lower bound?" in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science - ITCS '16*, pp. 357–368, Cambridge, MA, USA, January 2016.

[28] K. G. Larsen and J. B. Nielsen, "Yes, there is an oblivious RAM lower bound," in *Advances in Cryptology – CRYPTO 2018*, vol. 10992, pp. 523–542, Springer, Cham, 2018.

[29] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pp. 850–861, Denver, CO, USA, October 2015.

[30] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "PanORAMa: oblivious RAM with logarithmic overhead," in *Proceedings of the 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 871–882, Paris, France, October. 2018.

[31] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: optimal oblivious RAM," in *Proceedings of the Advances in Cryptology – EUROCRYPT 2020*, pp. 403–432, Springer, Zagreb, Croatia, May 2020.

[32] K. G. Larsen, M. Simkin, and K. Yeo, "Lower bounds for multi-server oblivious RAMs," in *Theory of Cryptography Conference*, vol. 12550, pp. 486–503, Springer, Cham, 2020.

[33] I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren, "Asymptotically tight bounds for composing ORAM with PIR," in *Public-Key Cryptography – PKC 2017*, vol. 10174, pp. 91–120, Springer, Berlin, Heidelberg, 2017.

[34] D. Cash, A. Drucker, and A. Hoover, "A lower bound for one-round oblivious RAM," in *Theory of Cryptography Conference*, vol. 12550, pp. 457–485, Springer, Cham, 2020.

[35] I. Komargodski and W.-K. Lin, "A logarithmic lower bound for oblivious RAM (for all parameters)," in *Advances in Cryptology – CRYPTO 2021*, vol. 12828, pp. 579–609, Springer, Cham, 2021.

[36] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly secure oblivious RAM without random oracles," in *Theory of Cryptography Conference*, vol. 6597, pp. 144–163, Springer, Berlin, Heidelberg, 2011.

[37] T.-H. H. Chan, K. Nayak, and E. Shi, "Perfectly secure oblivious parallel RAM," in *Theory of Cryptography Conference*, vol. 11240, pp. 636–668, Springer, Cham, 2018.

[38] M. Raskin and M. Simkin, "Perfectly secure oblivious RAM with sublinear bandwidth overhead," in *Proceedings of the Advances in Cryptology – ASIACRYPT 2019*, p. 27, Kobe, Japan, December 2019.

[39] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient private file retrieval by combining ORAM and PIR," in *Proceedings of the ISOC Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2014.

[40] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, "Verifiable oblivious storage," in *Public-Key Cryptography – PKC 2014*, vol. 8383, pp. 131–148, Springer, Berlin, Heidelberg, 2014.

[41] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: a constant bandwidth blowup oblivious ram," in *Theory of Cryptography Conference*, vol. 9563, pp. 145–174, Springer, Berlin, Heidelberg, 2016.

[42] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: achieving sublogarithmic read efficiency," in *Advances in Cryptology – CRYPTO 2018*, vol. 10991, pp. 371–406, Springer, Cham, 2018.