BULKOR: Enabling Bulk Loading for Path ORAM

Xiang Li
Tsinghua University
lixiang20@mails.tsinghua.edu.cn

Yunqian Luo
Tsinghua University
luoyq19@mails.tsinghua.edu.cn

Mingyu Gao
Tsinghua University
Shanghai AI Laboratory
Shanghai Qi Zhi Institute
gaomy@tsinghua.edu.cn

***Abstract*—Oblivious RAM (ORAM) is an important cryptographic primitive that aims to protect against data access pattern leakage. With the recent theoretical improvements in ORAM protocols and the introduction of hardware-based trusted execution environments (TEEs), ORAM has become an increasingly practical design that starts to be adopted in real-world secure systems. In this paper, we study the bulk loading problem of ORAM, i.e., constructing an ORAM structure with a large amount of data, which can benefit many scenarios in secure cloud systems, such as data recovery, layout conversion, and query processing. We propose BULKOR, an extension of the state-of-the-art Path ORAM protocol. BULKOR supports the deployment with TEEs in untrusted servers, and satisfies the doubly-oblivious requirement to alleviate the side channel concerns in modern TEEs. BULKOR improves both the theoretical complexity from $\mathcal{O}(N \log^3 N)$ to $\mathcal{O}(N \log^2 N)$, and the practical performance of ORAM bulk loading, without sacrificing the security guarantees. It significantly outperforms the baseline designs Oblix and ZeroTrace by $8.7\times$ to $54.6\times$ and $5.8\times$ to $533.1\times$, respectively, in various settings that implement ORAM on hard disks or in memory.**

1. Introduction

As data security increasingly gains attentions, secure database and data analytics systems quickly emerge in recent years. They support various kinds of privacy-preserving processing, from data storage outsourcing to encrypted query processing [1], [2], [3], [4]. However, although modern encryption and authentication techniques can protect data contents, advanced attacks can still be performed by the adversary to derive a significant amount of private information, by merely observing the access patterns to the sensitive data [5], causing severe security vulnerabilities.

Oblivious RAM (ORAM) protocols have been the standard and generic cryptographic solution to defend against this issue. It has been proposed [6] and improved [7], [8], [9], [10], [11] for decades, and now there exist practical designs that incur reasonable overheads compared to the non-oblivious setting. Currently, the most prevalent ORAM protocol is tree-based ORAM with $\mathcal{O}(\log N)$ bandwidth cost per access, where N denotes the total number of data blocks. Path ORAM [9], [12] is one such example. In the meantime, besides the theoretical improvements, hardware security techniques, in particular hardware-based trusted execution environments (TEEs), also are introduced to ORAM to reduce the networking communication cost between the remote client and the server [13]. A trusted domain known as the enclave can be instantiated at the untrusted server side as a local proxy. However, different from the fully trusted client, TEEs protect the data contents but not their access patterns. Double obliviousness [14] is required in such scenarios for both trusted and untrusted memory, which requires more complex ORAM controller designs inside TEEs. Nevertheless, with the progress from both the theoretical and system sides, ORAM techniques are becoming increasingly pragmatic, and have started to be applied to build oblivious database systems [14], [15].

In this paper, we motivate an opportunity to further improve ORAM systems in a previously less studied scenario — data bulk loading. Bulk loading is a well-known problem in database systems, which refers to loading a large amount of data and building a data structure, such as an index, from scratch as a whole in a short time [16], [17], [18], [19]. It is mostly used during system initialization, layout conversion, and data recovery. Although Oblix [14] proposed a bulk loading protocol, it was not significantly better than simply and serially performing write of each data block into ORAM in terms of both time complexity $\mathcal{O}(N \log^3 N)$ and practical performance. In this work, we aim to improve performance in both theory and practice without sacrificing security.

1.1. Motivation

We believe bulk loading for ORAM is an important topic, as there are many real-world scenarios that can benefit from a more efficient and secure bulk loading process. Below we describe several of such motivating use cases. We later formalize the problem in Section 2.3. We specifically target the scenarios where the server is equipped with a TEE and the clients would like to outsource all their private data to the server due to limited local storage.

Case 1: building block for oblivious algorithms. ORAM has been widely utilized in many applications as a basic building block for specially designed oblivious algorithms, such as database table join [20], stable match-

ing [21], [22], and breadth-first search [21], [23]. Although some designs [20] treat ORAM construction as a preprocessing step, when the oblivious algorithm needs to be performed on intermediate results (e.g., results of sub-queries), they would need to build ORAM structures on-the-fly, which is a major performance bottleneck. Therefore, it is important to explore the fast ORAM construction method, i.e., bulk loading, to reduce the end-to-end query processing latency. Cheap ORAM construction will also inspire more oblivious algorithm designs using ORAM, as we will demonstrate in more details in Section 6.3.

Case 2: data recovery. For the widely-used trusted proxy model of ORAM [24], [25], [26], [27], Vuppalapati et al. [28] pointed out that if the state of the centralized proxy is lost, the ORAM structure must be reconstructed on a new proxy, which involves downloading/uploading and decrypting/encrypting all the data and metadata. This would cause a long period of system unavailability. Since the TEE can be viewed as a trusted proxy located on an untrusted server that may crash, the aforementioned issue also exists in the TEE setting. If we could accelerate ORAM reconstruction, we can greatly reduce service unavailability and corresponding revenue loss.

Case 3: cloud storage services. Cloud storage services (e.g., Google Drive, Dropbox) allow users to store their data in cloud-side storage and also provide highly available access to these data. Such services are typically priced by storage capacity. Since an ORAM structure has larger size (e.g., $4\times$) than the non-oblivious layout, a more space-efficient approach to offering secure cloud storage services would be to store data in the non-oblivious layout at rest, and only convert into the ORAM structure when the user would like to perform oblivious accesses. Such a layout conversion should be fast enough to reduce user access delays. While converting an ORAM back to a simple layout is easy, e.g., through obliviously sorting all real and dummy blocks and dropping all dummy blocks, bulk loading data to build an ORAM requires specific optimizations. In addition to saving user expenses, this approach is also of interest to the cloud service providers to reduce their own infrastructure cost.

1.2. Our Contributions

In this paper, we investigate the bulk loading procedure for state-of-the-art tree-based ORAM protocols such as Path ORAM [9]. We mainly target scenarios with TEEs available on the untrusted server, considering their increasingly popular use and better efficiency. We consider their side-channel vulnerabilities and design our algorithms in a *doubly-oblivious* manner [14], i.e., both the access patterns within and outside the TEE are oblivious. We aim to improve both the *theoretical complexity* and the *practical performance* of ORAM bulk loading, without sacrificing any security guarantees. In other words, the bulk loaded ORAM should provide the same protection as the naive way of serial insertion [9, Section 3.4] for later normal accesses.

One key observation in our work is that, *a simple random shuffle of data to initialize the ORAM tree is insecure.* We therefore propose our efficient and secure construction, BULKOR, which assigns the path label of each data block in Path ORAM fully independently and randomly at the very beginning. Then BULKOR efficiently and effectively adjusts the actual location of each data block in the Path ORAM tree to eliminate any overflow issue, *without* changing the previously assigned path. The entire process is done obliviously, despite of the challenging task of moving data blocks in the irregular tree structure. Therefore, BULKOR achieves double obliviousness and can be used together with TEEs. We leverage existing oblivious sort primitives, and also propose two novel oblivious sub-procedures that are customized to our block adjustment tasks. BULKOR achieves $\mathcal{O}(N \log^2 N)$ time complexity using bitonic sort [29] or even $\mathcal{O}(N \log N)$ using bucket oblivious sort [30], improving upon the naive doubly-oblivious serial insertion which is $\mathcal{O}(N \log^3 N)$ with TEEs. BULKOR supports the standard Path ORAM recursion, so its space cost at the controller (in the enclave) remains the same as $\mathcal{O}(\log^2 N) \cdot \omega(1)$ [9]. The BULKOR implementation is also multi-threading parallelized to achieve good practical performance. Beyond Path ORAM, BULKOR can also be adapted to other tree-based ORAM protocols [10], [11], [31], [32], [33].

We implement BULKOR with Intel SGX, and evaluate its practical performance on different settings, including hard disks, in-memory, and fully in-enclave data storage, as well as different data block granularities from 64 B to 1 kB. We show that BULKOR is able to outperform the state-of-the-art Oblix [14] by $8.7\times$ to $54.6\times$ as the ORAM size increases. It also outperforms the ZeroTrace serial insertion baseline [13] by up to $160.6\times$ faster on hard disks, $493.6\times$ and $533.1\times$ with fully in-memory and in-enclave data, respectively. When applied to real-world application scenarios, BULKOR can accelerate various algorithms such as oblivious join, oblivious binary search, and oblivious BFS (case 1), improve the "9s" availability of database services by enabling faster data recovery (case 2), and save bills by roughly 50% for a cloud storage service using dynamically converted ORAM layouts (case 3). We have open sourced BULKOR at https://github.com/tsinghua-ideal/bulkor.

2. Preliminaries

2.1. Path ORAM

Path ORAM [9], [12] is a widely adopted ORAM protocol that offers superior performance in practice. The construction is split into two parts as the server and the client. Table 1 summarizes our notations. The server maintains the server storage in the form of a *binary tree* with $N-1$ nodes (indexed from 1 to N). Each tree node is a *bucket* that holds Z data blocks (e.g., $Z = 4$). Thus the total number of blocks is about $n = Z \cdot N$, among which up to N blocks are allowed to store encrypted real data, while the others are dummy blocks (also encrypted). We assign physical addresses to all blocks according to the breadth-first sequential representation (i.e., the heap format) of the ORAM tree. A block i belongs to the bucket α_i, and

TABLE 1. NOTATIONS.

Variable	Meaning
n	Total number of (data and dummy) blocks
N	Maximum number of data blocks
L	Height of ORAM tree, $L = \lceil \log_2 N \rceil - 1$
B	Block size
Z	Capacity of each bucket (in blocks)
a_i	Logical address of block i
p_i	Physical address of block i
α_i	Bucket in which block i is
x_i	Leaf label associated with block i
$P(x)$	Path from leaf x to tree root
S	Stash
position	Position map from logical address to leaf label
C_s	Stash capacity limit
C_p	Position map capacity in controller storage
$\mathbf{b} = \{b_i\}_{0 \le i < d}$	Input array of data blocks to be bulk loaded
$\mathbf{a} = \{a_i\}_{0 \le i < d}$	Input array of logical addresses
d	Length of input array (in blocks)

is located at the physical address $p_i \in [\alpha_i Z, (\alpha_i + 1)Z)$. For example, the root bucket contains blocks with physical addresses in $[Z, 2Z)$.

The client keeps the controller storage, including a *stash* that temporarily buffers data blocks, and a *position map*, which records the mapping from the logical address a_i of block i to the *leaf label* x_i, i.e., $x_i = \mathsf{position}(a_i)$. The *main invariant* of Path ORAM guarantees that a block must be either in the stash, or in one of the buckets along its corresponding root-to-leaf *path* $P(x_i) = \{x_i + N/2, x_i/2 + N/4, \ldots, 1\}$, where $x_i \in [0, N/2)$ is its leaf label and $x_i + N/2$ is the index of the leaf bucket.

To perform an access to a block at a_i, we first consult the position map to find its path $P(x_i)$ where $x_i = \mathsf{position}(a_i)$. For example in Figure 1, we access block 3 and fetch its path $P(0)$ from the ORAM tree. If this is the first write to a block (so the block does not exist yet), we randomly choose a leaf and its associated path. According to the main invariant, the target block must reside either in a bucket on the fetched path or in the stash, so we can now perform the desired read or write operation on it. In the meantime, we also reassign a new leaf label x_i' to the block through uniformly random sampling, and update $\mathsf{position}(a_i) = x_i'$ (in Figure 1 $\mathsf{position}(3)$ is updated from 0 to 3). Finally, we do *eviction* to minimize the number of blocks in the stash, avoiding stash overflow. We greedily push the data blocks in the stash or the buffer as far down the original path $P(x_i)$ as possible while constrained by the main invariant. More specifically, a block a_j can be put into a bucket along the path $P(x_i)$ if this bucket is also on its own path $P(x_j)$. In Figure 1, we see that block 6 can be pushed down to depth 2 since its leaf label is 0; block 7, whose leaf label is 1, can be pushed to depth 1. Now blocks 1 and 3 can be packed into the empty places at depth 0. In this case all the five data blocks are successfully evicted (with dummy blocks filling in the empty space). The updated path $P(x_i)$ is written back to the server ORAM tree.

Although the stash size is well bounded [9], [12], the

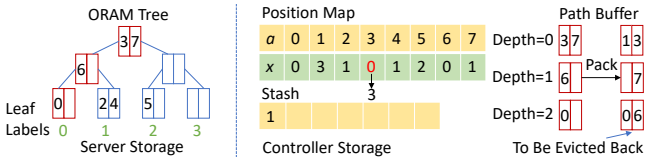

Figure 1. A toy example of path eviction in Path ORAM.

above design of Path ORAM requires a position map in the controller storage with a large $\mathcal{O}(N)$ space.[1] Path ORAM uses *recursion* to address this issue when |position| is larger than the capacity limit C_p [33], [34]. The position map itself is treated as data, stored and accessed in the same manner as a Path ORAM, with the help of another level of smaller position map. We recursively apply this technique until the position map gets small enough, e.g., $\mathcal{O}(1)$ size after $\mathcal{O}(\log N)$ recursions. Essentially, recursion trades access latency for controller storage.

2.2. Trusted Execution Environments

Trusted execution environments (TEEs), such as Intel SGX [35], [36], [37], Intel TDX [38], ARM TrustZone [39], and AMD SEV [40], create an efficient avenue for achieving data security. They split the hardware execution environment into a trusted domain (called an *enclave*) and an untrusted domain. The hardware guarantees the in-system isolation between the two domains, automatically encrypts and authenticates data when data leave the enclave, and ensures the legality of the code in the enclave through attestation [35].

However, existing TEEs are not a panacea. The security of TEEs has been questioned due to various side-channel vulnerabilities [41], [42], [43], [44], [45], [46], [47], [48], among which an important one is related to the leakage of access patterns, i.e., TEEs do not hide the address of each sensitive data access. Consequently, to ensure full security, even the accesses occurring inside the enclave would need additional protection, e.g., being oblivious.

2.3. Problem Definition and Threat Model

In this work we consider the ORAM bulk loading problem, where the client has securely outsourced its data to a remote server equipped with a TEE, and would like to run the bulk loading algorithm A_{load} to construct a new Path ORAM structure on the data, in preparation for subsequent accesses (denoted as A_{oram}) to the loaded Path ORAM.

Specifically, following the notations in Table 1, the input data of A_{load} are in the form of a block array $\mathbf{b}$ of length d, and each block is of size B. Generally, the blocks are not necessarily sorted in their logical addresses, i.e., each block b_i may have a logical address a_i not equal to its position i in the array. So we also take as input an array of the logical addresses, $\mathbf{a}$, of the same length d. The final

1. We consider constant-width addresses/labels, e.g., 64 bits, in typical system implementations; otherwise the space is $\mathcal{O}(N \log N)$.

output should be a valid Path ORAM structure, denoted as $R = (T, S, \mathsf{position})$, where T is a binary tree with $N-1$ nodes ($N \geq d$), each containing Z blocks, and S and $\mathsf{position}$ are a stash and a position map, respectively, which do not exceed their capacity limits C_s and C_p. For each data block b_i in $\mathbf{b}$, either there exists one and only one location in T that contains the block, or the block is in S. The position map should also contain consistent metadata, i.e., for each logical address a_i, the data block b_i resides on the path associated with the leaf label $x_i = \mathsf{position}(a_i)$. Overall, the bulk loading algorithm A_{load} constructs the expected ORAM structure R with a set of parameters: $R \leftarrow A_{\text{load}}(\mathbf{b}, \mathbf{a}, N, Z, C_s, C_p)$.

After executing A_{load}, the ORAM controller, including the position map and the stash, stays in the enclave. The enclave acts as a local trusted proxy of the client [13], [14], [15]. For each data access, only a single request and a single response are transmitted between the client and the server through networking; the heavy communication including path fetch and eviction happens within the local machine. This scenario also follows the proxy-based ORAM model [25], [27], [49], which can conveniently and efficiently handle multiple clients.

Threat model. The server is untrusted, potentially compromised by a malicious adversary, who may snoop, roll-back, or tamper with the sensitive data. Thus we need to ensure confidentiality, integrity, and freshness.

We assume the hardware processor of the server is trusted, while all software running on the processor is controlled by the adversary, except for the code inside the enclave, whose legality has been verified through attestation. The data values inside the enclave are protected, but the memory, disks, and networking are exposed to the adversary, who can observe the data stored/transferred on these devices as well as their corresponding addresses and access patterns. In other words, the enclave is not fully trusted, and suffers from access pattern side channels. Hence we aim to design a *doubly-oblivious* system [14], where the Path ORAM algorithm protects the external accesses to the server storage, and our oblivious implementation avoids leaking the internal access patterns of the enclave code. The security is hence nothing worse than the traditional client-server ORAM model.

Following common practice, we do not consider denial-of-service attacks, as well as physical attacks that exploit other side channels such as electromagnetic [46], thermal [47], and power [48].

3. Design Goals

In this work, we focus on improving the performance of Path ORAM bulk loading in the scenario with TEEs, while retaining the same security guarantees. We detail the performance and security goals in Sections 3.1 and 3.2. We show that such a construction is non-trivial by discussing a simple design in Section 3.3, which seems secure but actually violates some subtle requirements of ORAM.

3.1. Performance Goals

The most straightforward way to bulk load data blocks to a Path ORAM is to sequentially insert those blocks through the Path ORAM access protocol. Without loss of generality, we assume $d = N$. For each of the N ORAM accesses launched from the TEE, $\mathcal{O}(\log N)$ blocks (i.e., a path) are fetched/evicted from each of the $\mathcal{O}(\log N)$ recursive levels. The doubly oblivious requirement further restricts us to conduct full scans over the data structures in the TEE [13], [14], [15]. For example, when selecting candidate blocks to pack into the evicted path, we need to obliviously scan the stash, which brings an additional factor of $\mathcal{O}(\log N)$ [13]. Therefore the total cost increases to $\mathcal{O}(N \log^3 N)$.

To our best knowledge, Oblix [14] was the only existing (partial) solution but it omitted the position map construction details. To build an ORAM tree, Oblix assigned blocks to leaf buckets uniformly at random, followed by $\mathcal{O}(\log N)$ iterations to build the tree layer by layer, from bottom up. In each iteration, it first gathered the blocks assigned to the same buckets by oblivious sort. It then picked out the overflowed blocks (those not fitting in the bucket capacity Z) by another oblivious sort. These blocks were left to the next iteration, e.g., to be placed at the buckets of upper tree layers. Oblix still needed to pad each bucket with enough dummy blocks to hide the actual size. Overall, the time complexity was still $\mathcal{O}(N \log^3 N)$ despite optimization.

We aim to design a more efficient bulk loading algorithm to reduce the complexity, i.e., lower than $\mathcal{O}(N \log^3 N)$. Our algorithm does not increase the (trusted) controller storage, keeping it at $\mathcal{O}(\log^2 N) \cdot \omega(1)$ [9]. Besides the asymptotic improvements, our algorithm construction should also have practically high performance, with good locality and parallelism characteristics to leverage modern hardware.

3.2. Security Goals

Performance improvements are only valuable if we can retain the same level of security guarantees. We adopt a strong security model that follows all known binary-tree-based ORAM algorithms and their variants [33], [50], [51], and ensure that our bulk loading algorithm is statistically secure (excluding encryption [52], [53]). Furthermore, the same security requirements apply to the subsequent normal accesses after bulk loading. In other words, the final state of the Path ORAM must be compatible with a state that is reached through serial writes, i.e., for any subsequent accesses from this state, the externally observed trace is statistically independent. We illustrate a simple design that violates this requirement in Section 3.3.

To specify the leakage, we consider the following formalization. We use $\mathtt{params}$ to denote the system parameters (B, d, N, Z, C_s, C_p). Recall from Section 2.3 that the doubly-oblivious bulk loading algorithm A_{load} takes $\mathtt{params}$ and the arrays $\mathbf{b} = \{b_i\}$, $\mathbf{a} = \{a_i\}$ as inputs, and constructs the Path ORAM structure R; then subsequent accesses $\overrightarrow{y} = ((\text{op}_M, a_M, \text{data}_M), \ldots, (\text{op}_1, a_1, \text{data}_1))$ with a polynomial length M are performed on R using the Path ORAM

access algorithm A_{oram}. We define $\mathsf{Trace}_{\text{load}}(\mathsf{params}, \mathbf{b}, \mathbf{a})$ and $\mathsf{Trace}_{\text{oram}}(R, \overrightarrow{y})$ as the traces of these two processes, and let $\mathsf{Trace}(\mathsf{params}, \mathbf{b}, \mathbf{a}, \overrightarrow{y})$ be the concatenated trace $\mathsf{Trace}_{\text{load}}(\mathsf{params}, \mathbf{b}, \mathbf{a}) \parallel \mathsf{Trace}_{\text{oram}}(R, \overrightarrow{y})$.

Definition 1. *An ORAM built by bulk-loading is oblivious, if* $\forall \mathbf{a}$, $\forall \mathbf{b}$, *and* $\forall \overrightarrow{y}$, *there exists a probabilistic polynomial-time simulator* Sim *such that* $\mathsf{Sim}(\mathsf{params}, M) \cong \mathsf{Trace}(\mathsf{params}, \mathbf{b}, \mathbf{a}, \overrightarrow{y})$, *where* $\cong$ *means the two sides are statistically indistinguishable (excluding encryption).*

3.3. A Simple but Insecure Construction

A straightforward idea of bulk constructing a new Path ORAM would be to first randomly shuffle the blocks, treat the resultant layout as the ORAM tree, and accordingly build the position map afterwards. More specifically, we first pad the input array $\mathbf{b} = \{b_i\}$ to the length of $n = Z \cdot N$, which is the total number of blocks in the final ORAM tree. Then we randomly determine a permutation $\pi : [n] \to [n]$ (e.g., through format preserving encryption [54]), and shuffle the padded input array, i.e., for the new array $\mathbf{b}'$, $b'_{\pi(i)} = b_i$. The shuffled array is treated as the ORAM tree layout, i.e., the physical address $p_i = \pi(i)$ for block i. We next determine the associated leaf label x_i of each data block according to its p_i. This can be done by randomly choosing a leaf that belongs to the subtree under the tree node $\alpha_i = p_i/Z$ (the one in which p_i locates). This ensures that block i is on the path associated to leaf x_i. Note that a block with $p_i \in [0, Z)$ (bucket 0) can be seen as in the stash, so it can be associated with any leaf. Finally we build the position map as recursive levels of ORAM, using mappings of $a_i \to x_i$ for $i \in [0, N)$.

Such a bulk loading protocol is simple and efficient. However, it does not satisfy Definition 1. Consider two access sequences $\overrightarrow{y} = ((\text{read}, 0, \perp), (\text{read}, 1, \perp))$ and $\overrightarrow{y}' = ((\text{read}, 0, \perp), (\text{read}, 0, \perp))$ respectively performed just after finishing bulk loading. We abuse $P(\cdot)$ to denote the ORAM path for each access. Since π is a permutation, the physical addresses of blocks $p_i = \pi(i)$ and $p_j = \pi(j)$, $i \neq j$ will not be statistically independent. Hence for $\overrightarrow{y}$, the probability that both accesses have the same path can be calculated by categorizing whether the two blocks are (1) in the same bucket, (2) in different buckets but on the same path, or (3) on different paths. However for $\overrightarrow{y}'$, the leaf for block 0 is randomly re-sampled after the first access. So the second path for block 0 is independent of the first one. This makes the probability different from the above. We formally calculate and prove $\mathsf{Pr}[P(\overrightarrow{y}_0) = P(\overrightarrow{y}_1)] \neq \mathsf{Pr}[P(\overrightarrow{y}'_0) = P(\overrightarrow{y}'_1)]$ in Appendix A. This result means that we cannot construct one $\mathsf{Sim}(\mathsf{params}, 2)$ for two different distributions.

4. ORAM Bulk Loading with BULKOR

We propose BULKOR, an efficient and secure algorithm for ORAM bulk loading, achieving the performance and security goals in Section 3. Section 4.1 explains the BULKOR algorithm, with more details of its sub-procedures in Sections 4.2 to 4.4. We specifically illustrate the interaction with TEEs in Section 4.5. Section 4.6 analyzes the security guarantees of our algorithm. Finally, Section 4.7 extends BULKOR to support other tree-based ORAM protocols.

Algorithm 1: Path ORAM Bulk Loading

```
1  function BulkLoad(B, A, N, Z, C_s, C_p):
       Input: data block array B, logical address array A,
              number of buckets N, bucket capacity Z,
              stash capacity C_s, position map capacity C_p.
       Output: tree set T, stash set S, top-level position
               map P.
       /* Assign leaf label to block and initialize metadata array.
          Each element in M has following fields: original array
          index i, logical address a, leaf label x, bucket index α,
          new physical address in output tree p.                  */
2      M ← AssignLeaf(A);
       /* Recursively build position map.                          */
3      B_P, A_P, ← BuildPosMap(M, C_p);
4      if A_P is ∅ then
5          T, S, P ← ∅, ∅, B_P;
6      else
7          T, S, P ← BulkLoad(
                       B_P, A_P, |B_P|, Z, C_s, C_p);
       /* Compute the height of ORAM tree.                         */
8      L ← ⌈log_2 N⌉ − 1;
       /* Adjust bucket IDs to resolve bucket overflow.            */
9      OSort(M, _.α) ;                    // sort by bucket index
10     OAdjustBucketID(M, L, Z);
11     OSort(M, _.α) ;                    // sort by bucket index
       /* Assign physical addresses.                               */
12     OAssignPhyAddr(M);
       /* Place and fill in missing physical addresses.            */
13     OPlace(M, _.p) ;                   // place by physical address
14     OAssignPhyAddrDummy(M);
       /* Use metadata to place data blocks.                       */
15     OSort(M, _.i) ;                    // restore to original order
16     OSort((M, B), _.p) ;               // sort by new physical address
17     Split B and append to T, S;
18     if |S| > C_s then Abort ;
19     return T, S, P;
```

4.1. Overview

We summarize the overall algorithm of BULKOR in Algorithm 1. The key point to avoid the security issue in Section 3.3 is that, we must sample the leaf label of each data block *independently and uniformly at random*, to strictly confine with the Path ORAM definition [9]. Therefore, for each block i, we sample its leaf x_i from the range $[0, N/2)$ uniformly at random, and associate it with the block's logical address a_i to initialize the auxiliary metadata array (Line 2). These leaf assignments are *final*, and will not change in the later stages of bulk loading. We temporarily designate the block to the leaf bucket, i.e., $\alpha_i \leftarrow x_i + N/2$. These bucket assignments will be adjusted later due to bucket capacity overflow, in a way that does *not* change the

Algorithm 2: `BuildPosMap`

Input: metadata array $\mathcal{M}$, position map capacity C_p.
Output: data block array $\mathcal{B}_\mathcal{P}$, logical address array $\mathcal{A}_\mathcal{P}$, both for the position map itself.

1 **if** *current recursion depth* $= 0$ **then**
2 `OSort`($\mathcal{M}$, _.a) ; // sort by logical address
3 `OPlace`($\mathcal{M}$, _.a) ; // place by logical address
4 $\mathcal{A}_\mathcal{P} \leftarrow [\ ]$; $\mathcal{B}_\mathcal{P} \leftarrow [\ ]$;
5 $a \leftarrow 0$;
6 **foreach** $m \in \mathcal{M}$ **do**
7 $b \leftarrow m.x$;
8 Append a, b to $\mathcal{A}_\mathcal{P}$, $\mathcal{B}_\mathcal{P}$;
9 $a \leftarrow a + 1$;
10 **if** $|\mathcal{B}_\mathcal{P}| \leq C_p$ **then return** $\mathcal{B}_\mathcal{P}$, $\varnothing$;
11 **else return** $\mathcal{B}_\mathcal{P}$, $\mathcal{A}_\mathcal{P}$;

above finalized leaf assignments. The physical address p_i is left uninitialized. In addition, to hide data length and ensure obliviousness, we must pad the input blocks with dummy blocks to the full size of the resultant tree, i.e., $n = Z \cdot N$. All dummy blocks are initially designated to the dummy bucket 0 (the tree root starts at bucket 1).

The following design of BULKOR is quite different from and more efficient than Oblix [14]. With the finalized leaf assignments of all data blocks, we can now obliviously build the position map with all the leaf label mappings from the metadata array (Lines 3 to 7). Algorithm 2 shows the detailed process. First, we need to ensure the metadata are in the ascending order of logical addresses. This is done only once at the first recursion level, using the primitives of `OSort` and `OPlace` that will be explained shortly. Then, the leaf labels $\{x_i\}$ are treated as the data and form a new array, where each block may fit multiple leaf labels. If the position map capacity is within the limit, we finish the recursive build; otherwise we recursively invoke the bulk loading procedure. Figure 2 illustrates the result of the above Path ORAM recursion. There are three ORAM trees and a top-level, trivial position map in this case. The trivial position map and the first two trees together serve as the full position map, similar to multi-level page tables in operating systems. For each block in the tree at recursion level $l+1$, the leaves, which indicate paths in level l, are sorted by the associated logical addresses. Hence the leaf for level l can be directly indexed by $a_{i,l+1}$. Moreover, since the logical addresses are already in order, we omit the process of `OSort` and `OPlace`.

The core part of bulk loading is Lines 9 to 16 in Algorithm 1. To avoid the cost of moving large data blocks during layout adjustment, we primarily operate on the smaller metadata array, until their physical address assignments are finalized (Line 15). There are two key challenges that need to be addressed when determining the physical address of each block. First, as we designate all blocks to leaf buckets through independent random sampling, there would be non-negligible overflow happening. Therefore we need to *adjust the assigned buckets without changing the leaf labels*. This can be done by pushing the overflowed blocks up along their

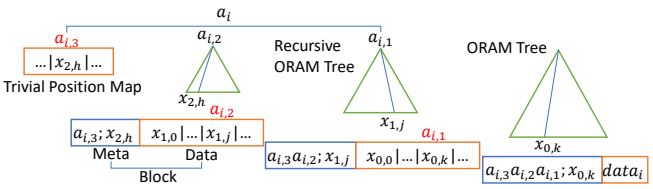

Figure 2. The recursive position map structure. The logical address a_i is split into several parts. The leaf label $x_{l,j}$ denotes a leaf belonging to the tree at level l. In a block at each level l, the metadata are the current logical address and the leaf label $x_{l,j}$; the data are several leaf labels for the previous level $\{x_{l-1,j}\}$.

paths to the tree root, but needs a *collaborative* way across all paths, and in a *doubly-oblivious* and *efficient* manner. Second, after we assign the final buckets, the placement of data blocks in the final tree must be done in a *doubly-oblivious* way. This is non-trivial considering the irregular bucket assignments resulted from the previous step. *There could be holes between the actual data blocks* in the ORAM tree layout, so simply sorting the blocks by their assigned physical positions will not work; dummy blocks must be placed in between. To overcome these issues, we propose two novel algorithms, `OAdjustBucketID` (Section 4.3) and `OPlace` (Section 4.4).

More specifically, in `OAdjustBucketID` (Line 10) that adjusts the bucket IDs, we first obliviously sort the metadata array by the current temporarily assigned bucket IDs. Then we scan the sorted array to check any violation of bucket overflow and adjust the block's bucket ID by pushing it to its parent. When the root bucket overflows, we move blocks to the stash space. At the end, we obliviously sort again to maintain the order by the new bucket IDs.

At this point, all buckets except the special dummy bucket 0 should satisfy the capacity constraint. Next, in `OAssignPhyAddr` (Line 12), we (obliviously) walk through the metadata array in the order of their bucket IDs, to assign the physical address to each block in the bucket. We only need $\mathcal{O}(1)$ auxiliary space. When encountering a new bucket ID, we set the block's physical address to the beginning of that bucket. If the bucket ID is the same as the last block, we increment the physical address by 1. The blocks in the stash also get contiguous physical addresses in the stash range. The dummy blocks in the dummy bucket 0 all obtain a dummy physical address 0 in this step.

With the physical address of each real data block, we then use `OPlace` (Line 13) to obliviously place them, where dummy blocks from the dummy bucket 0 fill the holes in between. Then we can fill in the physical address fields in all dummy block metadata *according to their current positions*, so that all physical addresses in the entire range are assigned to either real blocks or dummy blocks. All blocks in the stash are placed after those in the tree in this algorithm.

Finally, we use the metadata array to place the actual data blocks (Lines 15 to 16). We first restore the original order of the metadata array to pair them with the data array. Then we sort them in tandem by the new physical addresses. As the physical addresses of the stash are larger than those

of the tree, a simple split gives both the ORAM tree and the stash at the current recursion level, which are appended to the full sets $\mathcal{T}$ and $\mathcal{S}$. We prove that the resultant size of the stash $|\mathcal{S}|$ may only be larger than the capacity limit C_s with negligible probability (Section 4.3).

Complexity. We first analyze the non-recursive case. Most of the sub-procedures in Algorithm 1, including `AssignLeaf`, `BuildPosMap`, `OAssignPhyAddr`, and `OAssignPhyAddrDummy`, only need a scan on the input of $\mathcal{O}(N)$, with $\mathcal{O}(1)$ temporary space at the controller storage. In the later subsections we show that other procedures all have time complexity up to $\mathcal{O}(N \log^2 N)$, and need controller storage lower than $\mathcal{O}(\log^2 N) \cdot \omega(1)$. So the overall time complexity of BULKOR is $\mathcal{O}(N \log^2 N)$.

Further considering the recursive case, each recursion level needs a bulk loading. With typical parameters like 64 B blocks and 64-bit leaf labels, each block can contain $\chi \geq 2$ leaves. Therefore, the time complexity is $\mathcal{O}(N \cdot \log^2 N + N/\chi \cdot \log^2(N/\chi) + N/\chi^2 \cdot \log^2(N/\chi^2) + \cdots) = \mathcal{O}(\frac{N\chi}{\chi-1} \log^2 N)$.

It is also possible to reduce the time complexity down to $\mathcal{O}(N \log N)$ if using bucket oblivious sort [30] (Section 4.2). But we find the empirical performance becomes worse despite the better asymptotic complexity.

4.2. Choices of Oblivious Sort

The most frequently used building block in BULKOR is oblivious sort, which can be constructed from scratch or from oblivious shuffle [30]. There exist several practical, off-the-shelf designs. Some methods, e.g., Melbourne Shuffle [55] and Cache Shuffle [56], have low bandwidth cost $\mathcal{O}(N)$ with $\mathcal{O}(\sqrt{N})$ controller storage. However, they are not doubly oblivious and thus cannot be applied.

Bitonic sort [29] with time complexity $\mathcal{O}(N \log^2 N)$ is one of the most prevalent oblivious sort designs used in oblivious algorithms [15], [57]. It is also naturally doubly-oblivious for TEE scenarios. Another choice is bucket oblivious sort [30], the state-of-the-art $\mathcal{O}(N \log N)$ algorithm. To make the bucket oblivious sort algorithm doubly-oblivious, we need to make some adjustments to hide the access pattern in the trusted domain, i.e., by invoking bitonic sort in the MergeSplit step. To avoid notation confusion, we alias Z in bucket oblivious sort as θ [30]. The execution time is roughly $2N \log N \log^2 \theta$ with $\mathcal{O}(1)$ controller storage.

Although bucket oblivious sort has better asymptotic time complexity, to reduce the overflow probability to the standard 2^{-80}, we have to set $\theta = 512$, and the constant factor $2 \cdot \log^2 512 = 162$ in big $\mathcal{O}$ is large. Therefore, based on our empirical measurements, we prefer bitonic sort as the building block of BULKOR in practice, despite resulting in an overall time complexity of $\mathcal{O}(N \log^2 N)$.

4.3. Oblivious Bucket ID Adjustment

Recall that at the beginning of bulk loading, we randomly assign a leaf x_i to each block and temporarily put the block in its leaf bucket $\alpha_i \leftarrow x_i + N/2$. This may result in some leaf buckets having more than Z blocks and thus overflow. Hence we need to adjust the bucket ID of each overflow block to the parent bucket ID, by $\alpha_i \leftarrow \alpha_i/2$, so the block is still on its path, satisfying the main invariant of Path ORAM. If the parent is also full, we iteratively adjust the bucket ID to the grandparent until the block finds a empty slot. If no slot is found on the path until the root, the block is sent to the stash. If the stash is full, the algorithm fails (only with negligible chances).

Oblix [14] conducted such bucket ID adjustment by invoking oblivious sort for $\log N$ times (Section 3.1). We propose a better and novel algorithm to achieve the same effect, as in Algorithm 3. `OAdjustBucketID` modifies the bucket ID fields, α, of the metadata array. The input metadata array must be sorted by the initially assigned bucket IDs. $\alpha_i = 0$ is the dummy bucket for dummy blocks; $\alpha_i \geq N/2$ is a leaf bucket. When completed, the bucket IDs fall in the range $\alpha \in [0, N) \cup S$, where S is the stash range whose addresses are larger than any bucket IDs.

The core part of the algorithm uses an array of $L+1$ counters (L is the tree height) to record the occupancy of the buckets on the current path, i.e., how many real blocks are in those buckets. The algorithm sequentially visits the sorted initial bucket IDs of all blocks. For each new block,

Algorithm 3: `OAdjustBucketID`

Input: metadata array $\mathcal{M}$, height of ORAM tree L, bucket capacity Z.
Output: adjusted metadata array $\mathcal{M}$.

```
1  occ ← all 0s of length L + 1 ;                    // occupancy counters
2  lastid ← 0;
3  foreach m ∈ M do
       /* Find the first bit where bucket ID differs from the last,
          which decides the depth of the common ancestor anc. */
4      bitarr ← lastid ⊕ m.α; lastid ← m.α;
5      first ← true; anc ← −1;
6      for k ← 0 to L do
7          c ← bitarr[k] == 1 ∧ first;
8          cmov(c, first, false) ;                   // conditional move
9          cmov(c, anc, k);
10     cmov(first, anc, L + 1) ;                     // same as last bucket ID
       /* Reset path occupancy below common ancestor. */
11     for j ← 0 to L do
12         c ← j ≥ anc;
13         cmov(c, occ[j], 0);
       /* Adjust bucket ID to the first ancestor with free space. */
14     first ← true;
15     for j ← L to 0 do
16         c ← occ[j] < Z ∧ first;
17         cmov(c, first, false);
18         cmov(c, m.α, shift right m.α by L − j bits);
19         cmov(c, occ[j], occ[j] + 1);
       /* No free space along path; set to stash. */
20     cmov(first, m.α, S);
21 return M;
```

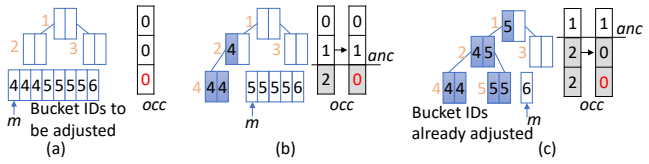

Figure 3. Oblivious bucket ID adjustment, with $N = 8$, $Z = 2$, $L = 2$. Padding (to 16) is omitted. Each bucket in the tree is marked by the orange bucket ID. The initially assigned bucket IDs of the blocks in (a) need adjustments. The blue background color means the block is finalized in that bucket and the initial bucket ID is changed to the orange number besides it. The gray background in the occupancy array denotes counter reset. The red occupancy number is the one to be incremented in this step.

it compares the bucket ID with the last block, and finds their common ancestor in the tree (Lines 4 to 10). The occupancy counters below the common ancester's depth are reset to 0, corresponding to the part of the path that changes (Lines 11 to 13). With the occupancy information, we can now try to push the block up along the path, until finding a bucket with free space (Lines 14 to 19). If there is no space, we use the stash (Line 20). The entire algorithm is oblivious by using the `cmov` (conditional move) assembly instruction and always iterating over the full range of each loop (Section 5).

We show an example in Figure 3. In (a) we show the initially assigned bucket IDs for the 8 data blocks. At the state of (b), we have processed the first three blocks that were all assigned to bucket 4 (the leftmost leaf), but one must be pushed up to bucket 2. The occupancy counters show the path 1-2-4, with 0, 1, and 2 blocks. Now we are looking at the first block that was assigned to bucket 5. The first different bit between the bucket IDs is the last one, so we reset the last counter to 0. Now the counters represent the path 1-2-5. We find that this block can stay at the empty bucket 5. Similarly, for (c) when we are at the last block, we reset the last two counters when changing path to 1-3-6.

Complexity. It is clear from Algorithm 3 that we iterate three loops of size $L = \mathcal{O}(\log N)$ for each block, so the total time complexity is $\mathcal{O}(N \log N)$, with the controller storage $\mathcal{O}(\log N)$ attributed to the occupancy counters.

Overflow probability. We have the following theorem.

Theorem 1. *With* `OAdjustBucketID` *in Algorithm 3, for* $Z \geq 2$*, there exist some constant* C_Z *and* α_Z *between 0 and 1, such that for sufficiently large* N*, the probability that the stash overflows (denoted by* $p_{overflow}$*) has an upper bound*

$$p_{overflow} < C_Z \alpha_Z^R \tag{1}$$

where R *is the size of the stash.*

In particular,

$$Z = 2, \quad p_{\text{overflow}} \leq 0.0407 \times 0.313^R \text{ (when } N > 2^{12}\text{)}$$
$$Z = 3, \quad p_{\text{overflow}} \leq 0.0076 \times 0.291^R \text{ (when } N > 2^{10}\text{)}$$
$$Z = 4, \quad p_{\text{overflow}} \leq 0.0021 \times 0.289^R \text{ (when } N > 2^{10}\text{)}$$
$$Z = 5, \quad p_{\text{overflow}} \leq 0.0021 \times 0.288^R \text{ (when } N > 2^{10}\text{)}$$

Recall that the overflow probability given in Path ORAM [9] is 14×0.6^R when $Z = 5$. Our method does not incur more failure risk than Path ORAM itself.

Proof sketch. The upper bound can be proved by estimating the moment-generating function (MGF) of the random variable X_{root}, which denotes the number of blocks pushed to the root. First, the MGF of a tree leaf is easy to compute, since it follows a binomial distribution. Second, for a non-leaf node, the random variables of block numbers pushed to its children are "negatively correlated". By strictly formalizing the negative correlation we then prove that the MGF of the sum of these two random variables is no more than the product of their MGFs. Now the MGF of X_{root} can be estimated by a recursion through the binary tree. The probability of overflow is finally upper bounded by choosing appropriate parameters in MGF and applying Markov's Inequality. The full proof is in Appendix B.

4.4. Oblivious Placement

As mentioned in Section 4.1, after assigning final physical addresses to blocks, a simple sort cannot place them to their correct positions in the tree layout due to the missing dummy blocks in between. We thus use our customized `OPlace` in Algorithm 4. `OPlace` is inspired by bitonic sort [29]. It requires the input array already sorted in the ascending order by the key. `OPlace` is invoked in two places in BULKOR. Algorithm 1 Line 13 uses the newly assigned physical address[2] as the key, to place blocks into their final positions. Algorithm 2 Line 3 uses the logical address as the key to process the input when building the position map.

Algorithm 4: `OPlace`

Input: sorted metadata array $\mathcal{M}$, key (address) field k.
Output: placed metadata array $\mathcal{M}$.
Data: Invalid key $\perp$ for unoccupied slot.

1 $n \leftarrow |\mathcal{M}|$; // Input should be padded to a power-of-2 size
2 $T = n/2$;
3 **for** $r \leftarrow 1$ **to** $\lfloor \log_2 n \rfloor - 1$ **do**
4 **for** $M \in \{0, 2T, 4T, \ldots, n - 2T\}$ **do**
5 $s = M + T$;
6 **for** $i \leftarrow M$ **to** $M + T - 1$ **do**
7 $j = i + T$;
8 $x \leftarrow \mathcal{M}[i].k$; $y \leftarrow \mathcal{M}[j].k$;
9 $c \leftarrow (x \neq \perp \wedge x \geq s) \vee (y \neq \perp \wedge y < s)$;
10 `oswap`$(c, \mathcal{M}[i], \mathcal{M}[j])$; // oblivious swap
11 $T \leftarrow T/2$;
12 **return** $\mathcal{M}$;

Figure 4 explains the algorithm with an example. In each stage with a new T, we obliviously swap each pair of out-of-order elements with a distance of T. After $\mathcal{O}(\log N)$ stages, all elements are in their desired positions.

2. Because now blocks are sorted by bucket IDs and `OAssignPhyAddr` assigns ascending physical addresses in each bucket, all blocks are already sorted by physical addresses.

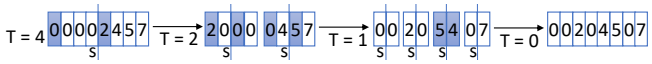

Figure 4. An example of oblivious placement. 0 denotes an invalid address. The blue background color denotes the blocks to be swapped in that stage.

Complexity. The algorithm runs in $\mathcal{O}(N \log N)$ time with $\mathcal{O}(1)$ controller storage.

Correctness. The full proof is in Appendix C. The proof is based on the observation that the algorithm is recursive. Each round of the main loop separates the array into two subarrays of half length. Each separated subarray has two interesting properties. First, each of its element has either an invalid key or a key that falls into the correct range of the subarray. Second, the set of valid keys are always in a contiguous range (if allowed to "rotate" around the subarray), and are strictly increasing. These two properties can be proved by induction. After the final round, all subarrays are of length 1, and now the desired result is a simple corollary of the first property.

4.5. Interaction with TEEs

We illustrate the interaction between BULKOR and TEEs, using Intel SGX as a representative TEE choice. Other TEEs can be used in a similar manner. The full procedure of Algorithm 1 runs inside the enclave. The data block and address arrays ($\mathcal{B}$ and $\mathcal{A}$) are loaded into the enclave and decrypted. N, Z, C_s, C_p are all public parameters. As the output, the initialized ORAM tree $\mathcal{T}$ is encrypted and stored outside, while the stash $\mathcal{S}$ and the top-level position map $\mathcal{P}$ stay inside the enclave in plaintext.

If the enclave memory is sufficiently large, e.g., with the newer SGX in Intel Scalable processor families [58], during the execution of Algorithm 1 the intermediate data can well fit within the enclave. Otherwise, if the enclave memory is limited, data would need to be encrypted, stored outside, and later loaded in and decrypted. To amortize repetitive enclave data swaps, such data loading happens in a batched manner, i.e., loading a large chunk of data each time. This is straightforward for streaming data accesses, e.g., `AssignLeaf`. However, `OPlace` and `OSort` have more complex patterns, for which we adopt customized batching methods. Specifically, for `OPlace` (Algorithm 4), the metadata array $\mathcal{M}$ may exceed the enclave memory size. We partition it into chunks and each time load two chunks whose elements need swaps. For example, the algorithm swaps $\mathcal{M}[i]$ and $\mathcal{M}[i+T]$ in each step (Line 10). We can load two chunks $\mathcal{M}[i : i+\text{len}]$ and $\mathcal{M}[i+T : i+T+\text{len}]$, swap the elements within the chunks, and then load the next two chunks. For `OSort` instantiated with bitonic sort, we adopt a similar manner.

Similar to the basic Path ORAM, BULKOR also naturally supports integrity and freshness protection against a malicious adversary (besides encrypting data). The basic unit of data transfers between the enclave and the external world is a chunk. We can authenticate accordingly, by appending a MAC to each chunk. For intermediate results that are only written/read as a whole through scans, we can simplify to use one checksum across the entire data array.

BULKOR involves multiple times of reads and writes of data bins, which is thus vulnerable to replay attacks. For example, in `OPlace`, a set of new chunks are generated in each round, acting as input to the next round. The adversary can replace them with stale chunks. Therefore, whenever a new set of data are generated, they are authenticated with a fresh random nonce, which is kept in the enclave for verification. The final ORAM tree is also protected with a mirrored Merkle tree [59], as the standard protocol [9].

4.6. Security Analysis

Theorem 2. *A doubly-oblivious Path ORAM built by our bulk loading algorithm* BULKOR *satisfies Definition 1.*

Proof. We need to construct a simulator `Sim` that generates the trace with only public information (i.e., the algorithm itself, and the system parameters including the input size), following the same distribution as the real trace.

In the bulk loading algorithm A_{load}, we observe that, (1) all loop boundaries and branch conditions are determined by public information, which is independent of the input sensitive data **b**, **a**; (2) conditional statements that must depend on the input are implemented through oblivious primitives (e.g., `cmov` instructions); (3) the memory operations (reads or writes) and accessed addresses in A_{load} are a fixed sequence determined by public information. Thus, `Sim` first generates random bits for the ciphertexts of inputs **b**, **a**. Then for each memory access, `Sim` also generates random bits of the same length as the encrypted content. `Sim` is able to generate these traces because the access pattern is deterministic.

For subsequent normal accesses A_{oram}, we rely on existing doubly-oblivious Path ORAM constructions like ZeroTrace [13] to make the manipulation on the stash and the trivial position map in the controller storage oblivious. As for $\texttt{Trace}_{\text{oram}}(R, \overrightarrow{y})$, we denote the access sequence seen by the adversary as $\mathsf{position}_M[a_M]$, $\mathsf{position}_{M-1}[a_{M-1}]$, $\ldots$, $\mathsf{position}_1[a_1]$. Notice that for any $\mathsf{position}_i[a_i], 0 < i \leq M$, it is either filled in Algorithm 2 with statistically independent values sampled uniformly at random, or reassigned randomly if the block has been accessed before. Therefore, `Sim` can just sample a sequence of positions uniformly at random and independently for each access, and generate random bits for the ciphertexts.

Finally `Sim` outputs the concatenation of both simulated traces. In this way, `Sim` generates an identical trace distribution as the real algorithm without the knowledge of the real input, so an adversary cannot extract any information about the input by eavesdropping on the memory traces. □

4.7. Supporting Other Tree-based ORAM

To our best knowledge, currently only Path ORAM has doubly-oblivious implementations [13], [14]. Nevertheless, we remark that BULKOR can be easily adapted to other tree-based ORAM schemes, as long as their doubly-oblivious designs for normal accesses are available. Note that tree-based

ORAMs [9], [10], [11], [31], [32], [33] typically maintain an ORAM tree in the server storage, and a position map and a stash in the controller storage. The main differences lie in the protocol sub-procedures and their metadata structures. These sub-procedures mainly influence the normal access procedure, and hence their differences are nearly orthogonal to ORAM initialization. Therefore, for different tree-based ORAM schemes, we only need to consider how to specifically initialize the corresponding metadata in the bulk loading procedure.

We give an example of how to extend BULKOR to Ring ORAM [10]. In Ring ORAM, each bucket contains $Z + D$ slots. The additional D slots are solely for dummy blocks, i.e., it is not allowed to store more than Z real blocks in each bucket. Therefore, the core part in Algorithm 1 (Lines 9 to 16) is performed with bucket size Z. After that, we run a scan-based sub-procedure to insert D additional dummy blocks in each bucket, and randomly and obliviously permute the blocks within each bucket. Ring ORAM also uses additional metadata for each bucket, e.g., `ptrs` indicate the in-bucket offsets for real blocks, `count` records the number of times the bucket has been touched, and `valids` indicate the validity of the $Z + D$ blocks [10]. They can all be filled in during the aforementioned scan. Specifically, `ptrs` are filled with the encrypted permuted locations, while `count` and `valids` are filled with 0 and 0/1 bits, respectively, only in authenticated plaintexts [10].

5. Implementation

We use Intel SGX [35], [36], [37], one of the most widely used TEE techniques. We implement BULKOR in Rust for high performance and memory safety, and we use Rust SGX SDK [60] for the enclave code. Our implementation consists of 7900 lines of code, in which about 5400 are enclave code that contributes to the trusted computing base (TCB). All basic oblivious functions, such as `cmov` and `oswap`, are implemented in assembly, mainly using the `CMOV` and `CMOVZ` x86 instructions, similar to ZeroTrace [13]. We also leverage Intel's AES-NI instructions to accelerate encryption/decryption. For leaf assignment and nonce generation, we use the true random number generator in SGX.

In-memory caching. Our implementation keeps the server storage in the external storage (e.g., hard disks) on the server. It is known that, manually caching the top of the ORAM tree in memory, instead of solely relying on OS page caches, could significantly improve performance [13]. We thus implement and extend the two-tier structure to a three-tier structure in BULKOR, i.e., enclave trusted memory, untrusted server memory, and disks. The top of the ORAM tree is cached in the enclave, free of costly encryption and authentication. The middle part is cached in untrusted memory, for faster accesses than the on-disk bottom part.

Parameters. The parameters involved in BULKOR are all configurable. For each block, we separately store its data and metadata, which are aligned with 64 B and 8 B, respectively. So the minimum size of a data block is 64 B. The choices for bucket capacity Z and stash size C_s are important. Larger values reduce overflow probability, but increase the cost of fetching/evicting a path and scanning the stash. If the position map exceeds its configured limit C_p, we use recursion. The capacity of the controller storage is configurable to allow BULKOR efficiently running in SGX enclaves with different sizes (e.g., 128 MB to 1 TB [58], [61]). We carefully manage the working set in the controller storage as described in Section 4.5. The configurable parameters also include the size constraints for the top and middle parts of the three-tier ORAM tree.

Parallelism and locality. We parallelize BULKOR with a multi-thread implementation for the major performance bottlenecks, oblivious sort and placement, which take over 95% of the total execution time. We follow the natural parallelization methods [62]. The other sub-procedures are currently sequential for ease of implementation, as they do not dominate overall execution. As for locality, steps like `OAdjustBucketID` only need sequential scans, exhibiting good locality. Moreover, when implementing on-disk ORAM, previous studies showed that when there are two disks, there exists a locality-friendly implementation of bitonic sort, which sorts n elements using $\mathcal{O}(n \log^2 n)$ bandwidth in only $\mathcal{O}(\log^2 n)$ disk seeks, denoted by $(2, \mathcal{O}(\log^2 n))$-locality [63]. We have not yet integrated this optimization in our current implementation and leave it for future work, as our test platform only has a single disk.

6. Evaluation

In this section, we empirically evaluate BULKOR with both micro-benchmarks and practical case studies.

6.1. Experimental Setup

Platform. To evaluate BULKOR, we use a server equipped with an Intel Xeon Gold 5317 processor of 3 GHz, 377 GB DDR4 memory, and external storage of a 2.4 TB, 10 kRPM, 12 Gbit/s hard disk (HDD). The processor supports SGX with a 64 GB enclave page cache (EPC). It runs Ubuntu 18.04 with Linux kernel 5.4.0. All experiments run on a single disk. We try to avoid the OS page cache impact, by using the command `nocache` to minimize the page cache effect. We also insert `sync_all` and `fadvise(POSIX_FADV_DONTNEED)` into the code, to make each write immediately flush to the disk and then clear the relevant page cache, isolating and better explaining the performance impact.

Parameters. For the ORAM data block size, we predominantly focus on 64 B (the minimum permitted size in our implementation) and 1 kB, corresponding to cacheline-level and page-level buckets [13]. The recursion ORAM block size is the same. We fix the bucket capacity $Z = 4$, which is a typical setting achieving balance between the stash overflow probability and the memory bandwidth cost [9]. It has been shown [9, Table 3] that when $Z = 4$, a stash size of $C_s = 89$ suffices for a negligible overflow probability 2^{-80}. We set the capacity limit of the trivial position map C_p to 4 kB.

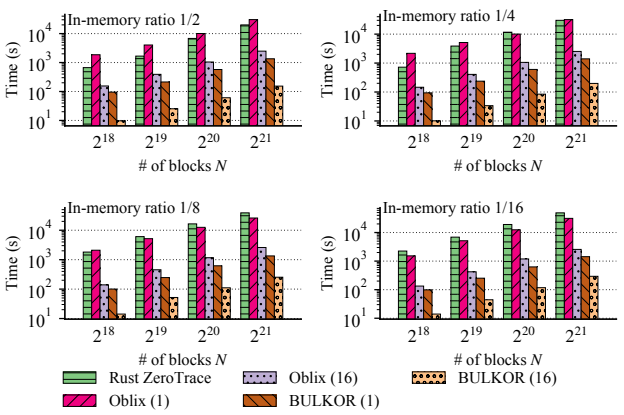

Figure 5. Comparison of overall performance on disks.

Moreover, the sizes of the top and middle tiers cached in memory greatly influence performance (Section 5). For fair comparisons, we set the size of the top tier (inside the enclave) equal to the controller storage consumed during BULKOR execution, which mainly holds the data chunks currently being processed by the multiple threads of BULKOR. We vary different size choices for the middle tier (in the server untrusted memory), but always ensure the same setting for both BULKOR and the baselines.

Baselines. We evaluate BULKOR for bulk loading N data blocks, against three baselines: the original C++ ZeroTrace [13], our re-implemented Rust ZeroTrace, and Oblix [14]. The first two baselines insert N blocks by performing N individual accesses to an initially empty Path ORAM. We implement the algorithm in Oblix in Rust by ourselves due to lack of code access. Same as ZeroTrace [13], we only measure the time for the controller (i.e., the enclave) to process data blocks, but omit the client-server communication to transfer queries or responses. We run each experiment for 10 times and use the average result. The cost of guaranteeing confidentiality and integrity is always included. Since BULKOR additionally supports multi-threading by design, we also evaluate this aspect. In all the figures, we use Oblix (x) and BULKOR (x) to represent running with x threads.

6.2. Benchmarking Overall Performance

We evaluate the overall performance of the four systems, C++ ZeroTrace, Rust ZeroTrace, Oblix, and BULKOR, on various cases. For page-sized 1 kB blocks, we assume that all blocks are stored on the disk, and only a portion of them can be cached in memory. We adjust the ratio for the in-memory part. For cacheline-sized 64 B blocks, we mainly test the in-memory case where the memory holds all the blocks. We only run C++ ZeroTrace in the fully in-memory scenarios due to the broken implementation for disks [64].

First, we investigate the performance of loading 1 kB data blocks on the disk, under different in-memory ratios, i.e., the ratio for how many blocks could be cached in memory. Figure 5 shows the execution time comparison. With an in-memory ratio of $1/16$, single-thread BULKOR achieves $22.8\times$ to $34.2\times$ speedups and $15.5\times$ to $21.9\times$ speedups over Rust ZeroTrace and Oblix, respectively. 16-thread BULKOR is $139.1\times$ to $160.6\times$ faster than Rust ZeroTrace, and $8.7\times$ to $10.1\times$ compared to 16-thread Oblix. As N gets larger, the speedup increases, which matches the trend of the asymptotic complexity improvement.

As the in-memory ratio increases, the performance of all systems improves due to fewer accesses to the slower disk. The speedups of BULKOR (and Oblix) over Rust ZeroTrace slightly decrease. BULKOR has more regular access patterns than those in Rust ZeroTrace (Section 5). Because the memory can better support random accesses than disks, the advantage of BULKOR becomes smaller for the in-memory case than that on disks. Hence, for in-memory ratios of $1/8$, $1/4$, and $1/2$, the average speedups of single-thread BULKOR over Rust ZeroTrace are $24.2\times$, $15.1\times$, and $9.9\times$; those of 16-thread BULKOR over Rust ZeroTrace are $135.7\times$, $114.5\times$, and $90.2\times$, respectively. The speedups of BULKOR over Oblix are rather stable across different ratios.

Next we evaluate bulk loading for both 64 B and 1 kB data blocks entirely in memory. As shown in Figures 6a and 6b, for both block sizes, as the number of blocks N grows, the execution time increases, and more importantly, the speedup of BULKOR improves following the algorithm complexity advantage (Section 4.1). We note that our re-implementation Rust ZeroTrace has similar performance to the original C++ ZeroTrace, so we mostly focus on comparing BULKOR with Rust ZeroTrace and Oblix. For 1 kB blocks, the speedup of single-thread BULKOR over Rust ZeroTrace ranges from $5.8\times$ to $8.8\times$, while using 16 threads increases the speedup to $21.3\times$ to $83.9\times$. The performance gains of BULKOR slightly decline compared to the on-disk settings. Again, this is because the random access overhead is reduced when all accesses occur in memory. The speedup of BULKOR over Oblix ranges from $11.4\times$ to $20.9\times$ for 1 thread and from $5.3\times$ to $17.8\times$ for 16 threads. For 64 B blocks, the speedup of single-thread BULKOR over Rust ZeroTrace ranges from $21.5\times$ to $54.2\times$, and multi-threading significantly improves the gains to $144.2\times$ to $493.6\times$. The speedup is higher than that of 1 kB blocks, since smaller blocks lead to more recursive levels and contribute a $\log N$ factor to ZeroTrace. Since BULKOR has more hardware-friendly sequential accesses, the speedups of BULKOR over Oblix are higher than those in 1 kB block settings, ranging from $28.3\times$ to $54.6\times$.

Figures 6c and 6d evaluate the cases when all data entirely fit in the enclave, for both 64 B and 1 kB data block sizes. We see slightly higher performance improvements of BULKOR over the baselines than the previous experiments which only cache 16 MB in the enclave. For 1 kB blocks, the speedups of single-thread and 16-thread BULKOR over Rust ZeroTrace are $6.2\times$ to $10.3\times$ and $30.3\times$ to $90.4\times$, respectively. For 64 B blocks, the corresponding speedups range from $21.3\times$ to $51.8\times$ (single-thread) and from $121\times$ to $533.1\times$ (16-thread). The main reason is that BULKOR needs repetitive encryption and decryption when swapping

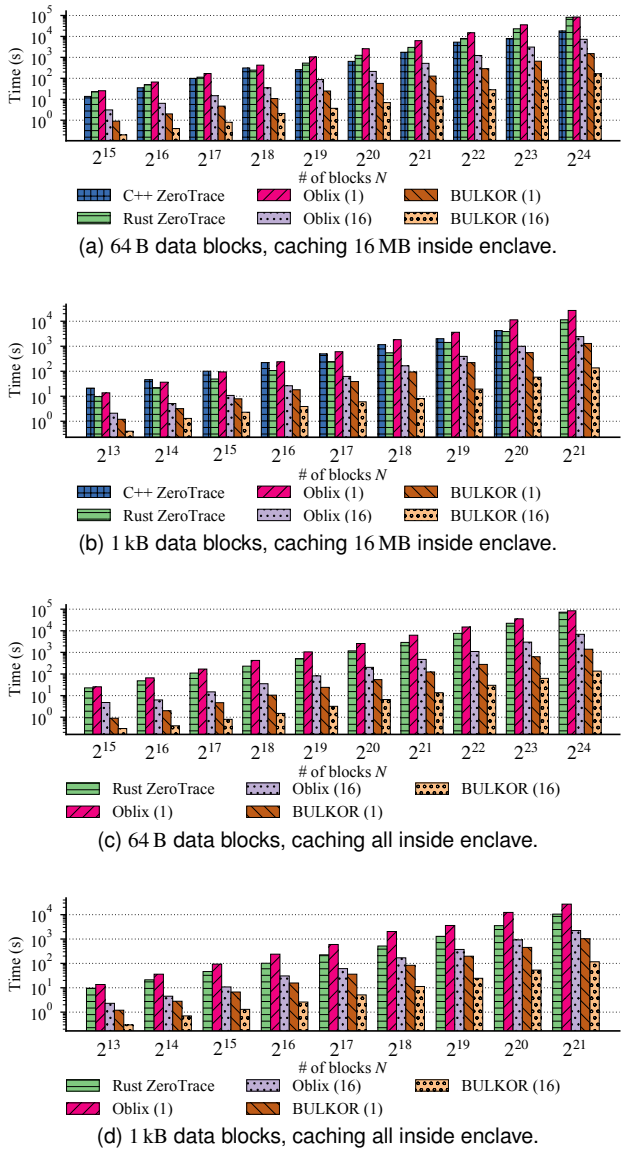

Figure 6. Comparison of overall performance in memory.

data out of the enclave, especially when performing bitonic sort and oblivious placement. Although ZeroTrace also needs to do so when fetching or evicting a path, it is not as frequent as in BULKOR. Caching all data inside enclaves alleviates these disadvantages of BULKOR over ZeroTrace.

Finally, we point out that, although currently BULKOR is not completely parallelized, the benefits of multi-threading are already significant because of the dominance of oblivious sort and placement. The performance of BULKOR can be further improved by adding more disks and optimizing locality as described in Section 5.

6.3. Practical Application Case Studies

We next evaluate BULKOR in the three application cases in Section 1.1, to show its potential usage and benefits.

Building block for oblivious algorithms. Certain oblivious algorithms have incorporated ORAM as a building block, and require constructing an ORAM structure on the latency-critical online phase (see Section 1.1 case 1). Here we show that BULKOR can be used to reduce end-to-end latencies of some algorithms and can also inspire more oblivious algorithm designs. In the following evaluations, we make all the data inside the TEE and use only 1 thread.

For oblivious join [20], we evaluate several sort-merge join benchmarks `SE1`-`SE3` [20] on social graphs [65]. BULKOR reduces the end-to-end latencies (including initialization) of the three benchmarks by 47.9%, 44.4%, and 42.8%, respectively. It means that initialization would take nearly half of the time if we cannot rely on preprocessing to build ORAM, e.g., when computing on intermediate results.

Next we consider oblivious search [21]. Apparently, for just a few search queries, it is better to use simple linear scan for obliviousness. Only when there are sufficient numbers of queries, the cost of building an ORAM to apply binary search can be amortized. So there exists a tradeoff point in terms of the number of queries, below which linear scan is preferred, and above which using ORAM is better. BULKOR is able to lower the tradeoff point by optimizing the initialization phase. In Table 2, we show the tradeoff points across different dataset sizes. With BULKOR, the tradeoff points greatly decrease, benefiting performance.

For oblivious breadth-first search (BFS), existing solutions [21], [23], [66] have used customized oblivious algorithms. The design in [66] (denoted as `DOGA`) does not use ORAM and is more suitable for dense graphs, with a time complexity $\mathcal{O}(V^2 \log V)$ for a graph with V vertices. `ObliBFS` [23] is ORAM-based and targets sparse graphs, with a time complexity $\mathcal{O}(VM \log^3 V)$.[3] However, it requires to know M, i.e., the upper bound of the number of adjacent nodes, which potentially leaks data information. `ObliBFS` also performs bad on dense graphs that have $M = \mathcal{O}(V^2)$. To support our claim that BULKOR could inspire more oblivious algorithms that use ORAM, we design a better doubly-oblivious BFS with $\mathcal{O}((V+E)\log^3(V+E))$ as described in Appendix D.[4] We evaluate these algorithms using real-world graphs [67]. We further apply BULKOR to our algorithm to improve the overall performance. As seen in Table 3, BULKOR improves the speedups of our algorithm over `DOGA` from the range 0.8×–9.1× to 1.2×–13.7×. Particularly, without BULKOR, our algorithm is slower than `DOGA` when the graph is dense (e.g., 4k vertices and 90k edges); BULKOR eliminates this degradation.

Data recovery. We then consider case 2 in Section 1.1, where an ORAM-based service fails and restarts. During data recovery, the service is in an unresponsive state. We compute the availability A using the mean time between failures (MTBF, from restart to next failure) and the mean time to recover (MTTR) [68], as $A = \text{MTBF}/(\text{MTBF}+\text{MTTR})$, assuming a single machine without replication. Typical MTTRs could be several hours [69], [70], while a common

3. [23] stated $\mathcal{O}(VM \log^2 V)$ without double obliviousness.
4. [21] mentioned an oblivious BFS but without algorithm details.

TABLE 2. THE TRADEOFF POINTS (IN TERMS OF SEARCH QUERY NUMBERS) ABOUT THE TWO APPROACHES FOR OBLIVIOUS SEARCH.

# of 64 B blocks	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
One search w/ linear scan (ms)	72	150	330	650	1200	2400	4900	9700
One search w/ ORAM (ms)	35	42	50	59	69	82	95	110
Tradeoff point w/o BULKOR	1.9×10^6	1.6×10^6	1.5×10^6	1.7×10^6	2.0×10^6	2.4×10^6	2.7×10^6	3.1×10^6
Tradeoff point w/ BULKOR	3.2×10^3	2.4×10^3	2.1×10^3	2.1×10^3	2.4×10^3	2.5×10^3	2.6×10^3	2.8×10^3

TABLE 3. OBLIVIOUS BFS PERFORMANCE (IN SECONDS).

Graph: (V,E)	(4k,90k)	(5k,15k)	(6k,21k)	(8k,26k)
DOGA	136	379	461	622
ObliBFS	2536	302	434	587
Ours w/o BULKOR	166	41	57	72
Ours w/ BULKOR	110	28	38	48

specification for PATA and SATA drives has an MTBF of 300,000 hours [71]. For hundreds of GB of data, BULKOR would reduce the bulk loading overheads from hundreds of hours to several hours. Expressed as the class of "9s" defined by $c = \lfloor -\log_{10}(1-A) \rfloor$, the improvement of BULKOR over the baseline is about 2 "9s", determined by the two orders of magnitude speedup demonstrated in Section 6.2.

Cloud storage services. Assume a user outsources 20 GB data to a cloud storage. During the day, data are stored in the ORAM format; at night, data are condensed into normal encryption to save space (and thus cloud bills). Such daily data conversion must be fast enough to realize the benefits, e.g., within an hour compared to every 24 hours. In the disk setting of 1 kB data blocks and $Z = 4$, it takes around $4{\times}10^5$ s for Rust ZeroTrace to do such a conversion, impossible to achieve the goal, while BULKOR spends less than an hour. With the enabled $4\times$ space saving, the user could save the expense by roughly $1 - \frac{8\,\text{h}\times 4 + 16\,\text{h}\times 1}{24\,\text{h}\times 4} = 50\%$.

7. Related Work

This work motivates the problem of bulk loading for tree-based ORAM protocols, in particular Path ORAM [9]. The tree-based ORAM protocol was first proposed by [33], and later designs [9], [10], [32] extended upon the original construction. All these protocols were statistically secure, had rather simple concepts, and exhibited practical performance results. They mostly focused on optimizing the bandwidth cost and the client storage size from a theoretical perspective. Our work takes a system's perspective to consider the diverse real-world application scenarios.

The introduction of TEEs [35], [36], [37], [39], [40] further enlarges the use cases of ORAM, as it allows to instantiate a trusted enclave closer to the untrusted storage, greatly reducing the network communication cost. However, it may suffer from more side-channel vulnerabilities than the traditional remote client-server model. ZeroTrace [13] was the first to combine tree-based ORAMs (Path ORAM and Circuit ORAM) with Intel SGX [35], [36], [37]. It designed oblivious memory primitives to avoid exposing the access patterns of the code inside enclaves. Oblix [14] conceptualized the *doubly-oblivious* requirement for both the implementations inside enclaves and the original external accesses. It designed higher-level oblivious data structures, e.g., (multi-)maps. While Oblix tried to serve transaction processing (i.e., OLTP), Opaque [57] focused on data analytics (i.e., OLAP). It designed several oblivious operators, such as join and aggregation. The key building block in Opaque was also oblivious sort. ObliDB [15] designed a set of new oblivious query processing algorithms and further supported more general query workloads. It carefully selected between two storage methods for database tables according to the query selectivities. For high selectivities, queries were served by an oblivious B+tree, similarly as in Oblix [14]; for low selectivities, they were served in the flat storage, typically with several rounds of scans. It aimed to support both OLTP and OLAP. Our proposed algorithm, BULKOR, can be readily applied to them to further optimize the initialization phase and data recovery through oblivious bulk loading.

8. Conclusions

We implemented BULKOR, an optimized Path ORAM bulk loading algorithm that supports double obliviousness when incorporating hardware enclaves at the untrusted server side. BULKOR achieves $\mathcal{O}(N \log^2 N)$ time complexity, improving upon the naive serial insertion, with the same space cost at the controller. When practically measured, BULKOR also outperforms the baseline by two orders of magnitude. Given its efficiency and security, BULKOR could be useful in various real-world cases.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262) and the Institute for Interdisciplinary Information Core Technology, Xi'an. Mingyu Gao is the corresponding author.

References

[1] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu, "Two Can Keep a Secret: A Distributed Architecture for Secure Database Services," in *CIDR*, 2005, pp. 186–199.

[2] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *SOSP*, 2011, pp. 85–100.

[3] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan, "Secure Database-as-a-Service with Cipherbase," in *SIGMOD*, 2013, pp. 1033–1036.

[4] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database Using SGX," in *S&P*, 2018, pp. 264–278.

[5] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *NDSS*, 2012.

[6] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *STOC*, 1987, pp. 182–194.

[7] R. Ostrovsky, "Efficient Computation on Oblivious RAMs," in *STOC*, 1990, pp. 514–523.

[8] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *JACM*, vol. 43, no. 3, pp. 431–473, 1996.

[9] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *JACM*, vol. 65, no. 4, pp. 1–26, 2018.

[10] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants Count: Practical Improvements to Oblivious RAM," in *USENIX Security*, 2015, pp. 415–430.

[11] C. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov, "Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM," Cryptology ePrint Archive, Paper 2015/1065, 2015.

[12] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," in *CCS*, 2013, p. 299–310.

[13] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace : Oblivious Memory Primitives from Intel SGX," in *NDSS*, 2018.

[14] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An Efficient Oblivious Search Index," in *S&P*, 2018, pp. 279–296.

[15] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious Query Processing for Secure Databases," *VLDB*, vol. 13, no. 2, p. 169–183, 2019.

[16] S. Berchtold, C. Böhm, and H.-P. Kriegel, "Improving the Query Performance of High-Dimensional Index Structures by Bulk Load Operations," in *EDBT*, 1998, pp. 216–230.

[17] M. Nicola and J. John, "XML Parsing: A Threat to Database Performance," in *CIKM*, 2003, pp. 175–178.

[18] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The Vertica Analytic Database: C-Store 7 Years Later," *VLDB*, vol. 5, no. 12, p. 1790–1801, 2012.

[19] A. Papadopoulos and Y. Manolopoulos, "Parallel Bulk-Loading of Spatial Data," *Parallel Computing*, vol. 29, no. 10, pp. 1419–1444, 2003.

[20] Z. Chang, D. Xie, S. Wang, and F. Li, "Towards Practical Oblivious Join," in *SIGMOD*, 2022, p. 803–817.

[21] S. Zahur, X. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation," in *S&P*, 2016, pp. 218–234.

[22] J. Doerner, D. Evans, and a. shelat, "Secure Stable Matching at Scale," in *CCS*, 2016, p. 1602–1613.

[23] P. Moreno-Sanchez, A. Kate, M. Maffei, and K. Pecina, "Privacy Preserving Payments in Credit Networks," in *NDSS*, 2015.

[24] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: A Parallel Oblivious File System," in *CCS*, 2012, pp. 977–988.

[25] E. Stefanov and E. Shi, "ObliviStore: High Performance Oblivious Cloud Storage," in *S&P*, 2013, pp. 253–267.

[26] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi, "Obladi: Oblivious Serializable Transactions in the Cloud," in *OSDI*, 2018, pp. 727–743.

[27] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "TaoStore: Overcoming Asynchronicity in Oblivious Data Storage," in *S&P*, 2016, pp. 198–217.

[28] M. Vuppalapati, K. Babel, A. Khandelwal, and R. Agarwal, "SHORTSTACK: Distributed, Fault-tolerant, Oblivious Data Access," in *OSDI*, 2022, pp. 719–734.

[29] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Transactions on Computers*, vol. 28, no. 1, pp. 2–7, 1979.

[30] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket Oblivious Sort: An Extremely Simple Oblivious Sort," in *SOSA*, 2020, pp. 8–14.

[31] K.-M. Chung, Z. Liu, and R. Pass, "Statistically-Secure ORAM with $\tilde{O}(\log^2 n)$ Overhead," in *ASIACRYPT*, 2014, pp. 62–81.

[32] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *CCS*, 2015, pp. 850–861.

[33] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost," in *ASIACRYPT*, 2011, pp. 197–214.

[34] E. Stefanov, E. Shi, and D. X. Song, "Towards Practical Oblivious RAM," in *NDSS*, 2012.

[35] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," *HASP@ISCA*, 2013.

[36] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," *HASP@ISCA*, 2013.

[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," *HASP@ISCA*, 2013.

[38] Intel, "Intel TDX," https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html, 2020.

[39] T. Alves, "Trustzone: Integrated Hardware and Software Security," *White paper*, 2004.

[40] D. Kaplan, "Protecting VM Register State with SEV-ES," *White paper*, 2017.

[41] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *EuroSec*, 2017, pp. 1–6.

[42] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security*, 2017, pp. 557–574.

[43] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017, pp. 3–24.

[44] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *USENIX Security*, 2017, pp. 1041–1056.

[45] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, 2015, pp. 640–656.

[46] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic Analysis: Concrete Results," in *CHES*, 2001, pp. 251–261.

[47] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, "Thermal Covert Channels on Multi-Core Platforms," in *USENIX Security*, 2015, pp. 865–880.

[48] M. Zhao and G. E. Suh, "FPGA-Based Remote Power Side-Channel Attacks," in *S&P*, 2018, pp. 229–244.

[49] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang, "Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward," in *CCS*, 2015, pp. 837–849.

[50] K.-M. Chung and R. Pass, "A Simple ORAM," Cornell University, Tech. Rep., 2013.

[51] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing ORAM and Using it Efficiently for Secure Computation," in *PETS*, 2013, pp. 1–18.

[52] M. Ajtai, "Oblivious RAMs without Cryptographic Assumptions," in *STOC*, 2010, pp. 181–190.

[53] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly Secure Oblivious RAM without Random Oracles," in *TCC*, 2011, pp. 144–163.

[54] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption," *NIST Special Publication*, 2016.

[55] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The Melbourne Shuffle: Improving Oblivious Storage in the Cloud," in *ICALP*, 2014, pp. 556–567.

[56] S. Patel, G. Persiano, and K. Yeo, "CacheShuffle: A Family of Oblivious Shuffles," in *ICALP*, 2018.

[57] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *NSDI*, 2017, pp. 283–298.

[58] Intel, "3rd Gen Intel Xeon Scalable Processors with Intel SGX," https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html, 2021.

[59] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *CRYPTO*, 1987, pp. 369–378.

[60] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards Memory Safe Enclave Programming with Rust-SGX," in *CCS*, 2019, pp. 2333–2350.

[61] Intel, "10th Gen Intel Core Processor Families Datasheet, Vol. 1," https://www.intel.com/content/www/us/en/products/docs/processors/core/10th-gen-core-families-datasheet-vol-1.html, 2020.

[62] M. F. Ionescu and K. E. Schauser, "Optimizing Parallel Bitonic Sort," in *IPPS*, 1997, pp. 303–309.

[63] G. Asharov, T.-H. Hubert Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Locality-Preserving Oblivious RAM," in *EUROCRYPT*, 2019, pp. 214–243.

[64] Sajin Sasy, "Notes on ZeroTrace," https://github.com/sshsshy/ZeroTrace#other-notes, 2022.

[65] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring User Influence in Twitter: The Million Follower Fallacy," in *ICWSM*, 2010.

[66] M. Blanton, A. Steele, and M. Alisagari, "Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing," in *ASIA CCS*, 2013, p. 207–218.

[67] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[68] J. Lienig and H. Bruemmer, "Reliability Analysis," in *Fundamentals of Electronic Systems Design*, 2017, pp. 45–73.

[69] I. Aziz, S. Karim, and M. Hossain, "Effective Implementation of Total Productive Maintenance and Impacts on Breakdown Time and Repair & Maintenance — A Case Study of a Printing Industry in Bangladesh," in *Proceedings of the Global Engineering, Science and Technology Conference*, 2012, pp. 1–9.

[70] I. Ribeiro, R. Godina, C. Pimentel, F. Silva, and J. Matias, "Implementing TPM Supported by 5S to Improve the Availability of an Automotive Production Line," *Procedia Manufacturing*, vol. 38, pp. 1574–1581, 2019.

[71] Wikipedia, "Annualized failure rate — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Annualized%20failure%20rate&oldid=1024295897, 2023, [Online; accessed 28-July-2023].

Appendix A. Probability Calculation of Section 3.3

For $\overrightarrow{y}$, the probability that both accesses have the same path can be calculated by categorizing whether two blocks are in the same bucket, or in different buckets but on the same path, or on different paths. We treat the bottom layer as the 0th layer, and there are in total L layers. The probability that two blocks are assigned to a specific path and put into the same bucket in the ith layer is $f_0(i) = \frac{Z}{ZN}\frac{1}{2^i} \cdot \frac{Z-1}{ZN-1}\frac{1}{2^i}$, while the probability that one block is put into the bucket in the ith layer and the other block is at a bucket in other layers is $f_1(i) = \frac{Z}{ZN}\frac{1}{2^i} \cdot \frac{Z}{ZN-1}(1-\frac{1}{2^i})$. Let $g(i) = f_0(i) + f_1(i)$ be the probability that the two blocks are on the same path in the ith layer. Then, for all the $N/2$ paths and across all L layers, we have $\mathtt{Pr}[P(\overrightarrow{y}_0) = P(\overrightarrow{y}_1)] = \frac{N}{2}\left(\sum_{i=0}^{L} g\left(i\right) + g\left(L\right)\right)$, where the second term is for the stash space.

However for $\overrightarrow{y}'$, the leaf for block 0 is randomly resampled after the first access. So the second path for block 0 is independent of the first one. Similar to the above, the probability that block 0 is initialized to a specific path in the ith layer is $\frac{Z}{ZN}\frac{1}{2^i}$, and the probability that block 0 is assigned to the same path by the normal Path ORAM algorithm is simply $2/N$. Therefore, $\mathtt{Pr}[P(\overrightarrow{y}'_0) = P(\overrightarrow{y}'_1)] = \frac{N}{2}\frac{Z}{ZN}\left(\sum_{i=0}^{L}\frac{1}{2^i} + \frac{1}{2^L}\right)\frac{2}{N} = \frac{2}{N}$.

So $\mathtt{Pr}[P(\overrightarrow{y}_0) = P(\overrightarrow{y}_1)] \neq \mathtt{Pr}[P(\overrightarrow{y}'_0) = P(\overrightarrow{y}'_1)]$.

Appendix B. Proof of Overflow Probability

Let T be the ORAM tree. T is a perfect binary tree of height L, thus having $N = 2^{L+1} - 1$ nodes and $n = 2^L$ leaves. For each node u in T, let X_u be the random variable representing the maximum number of blocks it contains during the process. Let $Y_u = \max\{X_u - Z, 0\}$ for any u.

Lemma 1. *For every node u in T, if p and q are its children, then for all $t \in \mathbb{R}$,*

$$\mathbb{E}\left[\exp\left(tY_p\right)\right] \cdot \mathbb{E}\left[\exp\left(tY_q\right)\right] \geq \mathbb{E}\left[\exp\left(t(Y_p + Y_q)\right)\right] \quad (2)$$

Proof. For every node u in T, let B_u be the total number of blocks in the subtree rooting at u. Then $B_u = B_p + B_q$. Given b and t, for any $\beta \geq 0$, let $r_\beta = \mathbb{E}\left[\exp(tY_p) \mid B_p = \beta\right]$, $s_\beta = \mathbb{E}\left[\exp(tY_q) \mid B_q = b - \beta\right]$, and $k_\beta = \mathbb{E}\left[B_p = \beta\right] = \mathbb{E}\left[B_q = \beta\right]$ (the second equality comes from symmetry). Since each block is placed independently, r_β is non-decreasing with β, and s_β is non-increasing with β.

Now for any $b \geq 0$,

$$\begin{aligned} &\mathbb{E}\left[\exp(t(Y_p + Y_q)) \mid B_u = b\right] && (3)\\ &= \mathbb{E}\left[\exp(tY_p) \cdot \exp(tY_q) \mid B_p + B_q = b\right] && (4)\\ &= \sum_{\beta=0}^{b} \mathbb{E}\left[B_p = \beta\right] \end{aligned}$$

$$\cdot \mathbb{E}\left[\exp(tY_p)\cdot\exp(tY_q) \mid B_p=\beta, B_q=b-\beta\right] \quad (5)$$

$$= \sum_{\beta=0}^{b} \mathbb{E}\left[B_p=\beta\right]\cdot\mathbb{E}\left[\exp(tY_p) \mid B_p=\beta, B_q=b-\beta\right] \cdot \mathbb{E}\left[\exp(tY_q) \mid B_p=\beta, B_q=b-\beta\right] \quad (6)$$

$$= \sum_{\beta=0}^{b} \mathbb{E}\left[B_p=\beta\right]\cdot\mathbb{E}\left[\exp(tY_p) \mid B_p=\beta\right] \cdot \mathbb{E}\left[\exp(tY_q) \mid B_q=b-\beta\right] \quad (7)$$

$$= \sum_{\beta=0}^{b} k_\beta r_\beta s_\beta \quad (8)$$

where the equality relations in (6) and (7) come from the fact that given the number of blocks in a subtree, its distribution of blocks is independent of the blocks in the rest of tree, since all blocks are placed independently.

In addition

$$\mathbb{E}\left[\exp(tY_p) \mid B_u=b\right] \quad (9)$$

$$= \sum_{\beta=0}^{b} \mathbb{E}\left[B_p=\beta\right]\cdot\mathbb{E}\left[\exp(tY_p) \mid B_p=\beta\right] \quad (10)$$

$$= \sum_{\beta=0}^{b} k_\beta r_\beta \quad (11)$$

Similarly

$$\mathbb{E}\left[\exp(tY_q) \mid B_u=b\right] = \sum_{\beta=0}^{b} k_\beta s_\beta \quad (12)$$

Hence

$$\mathbb{E}\left[\exp(t(Y_p+Y_q)) \mid B_u=b\right] - \mathbb{E}\left[\exp(tY_p) \mid B_u=b\right]\cdot\mathbb{E}\left[\exp(tY_q) \mid B_u=b\right] \quad (13)$$

$$= \sum_{\beta=0}^{b} k_\beta r_\beta s_\beta - \sum_{\beta=0}^{b} k_\beta r_\beta \cdot \sum_{\beta=0}^{b} k_\beta s_\beta \quad (14)$$

$$= \sum_{\beta=0}^{b} k_\beta r_\beta s_\beta \sum_{\beta=0}^{b} k_\beta - \sum_{\beta=0}^{b} k_\beta r_\beta \cdot \sum_{\beta=0}^{b} k_\beta s_\beta \quad (15)$$

$$= \frac{1}{2}\sum_{\beta_1=0}^{b}\sum_{\beta_2=0}^{b} k_{\beta_1}k_{\beta_2}(r_{\beta_1}-r_{\beta_2})(s_{\beta_1}-s_{\beta_2}) \quad (16)$$

$$\le 0 \quad (17)$$

Since the above inequality holds for any b, we conclude that

$$\mathbb{E}\left[\exp(t(Y_p+Y_q))\right] - \mathbb{E}\left[\exp(tY_p)\right]\cdot\mathbb{E}\left[\exp(tY_q)\right] \le 0 \quad (18)$$

Thus we complete the proof of the lemma. □

For a leaf c in T, since X_c follows a $(1/n, nZ)$-Bernoulli distribution, we have

$$\mathbb{E}\left[\exp\left(tX_c\right)\right] = \left(1+\frac{1}{n}(e^t-1)\right)^{nZ} \quad (19)$$

Then if u has two children p, q, we have

$$X_u = \max\{p-Z,0\} + \max\{q-Z,0\} = Y_p + Y_q \quad (20)$$

Then for any $t>0$,

$$\mathbb{E}\left[\exp\left(tX_u\right)\right] \quad (21)$$

$$= \mathbb{E}\left[\exp\left(t(Y_p+Y_q)\right)\right] \quad (22)$$

$$\le \mathbb{E}\left[\exp\left(tY_p\right)\right]\cdot\mathbb{E}\left[\exp\left(tY_q\right)\right] \quad (23)$$

$$= \mathbb{E}\left[\exp\left(t\max\{X_p-Z,0\}\right)\right] \cdot \mathbb{E}\left[\exp\left(t\max\{X_p-Z,0\}\right)\right] \quad (24)$$

$$\le \left(1+e^{-Zt}\mathbb{E}\left[\exp\left(tX_p\right)\right]\right) \cdot \left(1+e^{-Zt}\mathbb{E}\left[\exp\left(tX_q\right)\right]\right) \quad (25)$$

Lemma 2. *For $0<p<1/4$, the sequence $\{a_n\}_{n\ge0}$ defined by $a_{n+1}=(1+pa_n)^2$ converges to $\frac{2}{1-2p+\sqrt{1-4p}}$ if $0<a_0<\frac{1-2p+\sqrt{1-4p}}{2p^2}$.*

Proof. Notice that

$$\alpha = \frac{2}{1-2p+\sqrt{1-4p}} \quad (26)$$

$$\beta = \frac{1-2p+\sqrt{1-4p}}{2p^2} \quad (27)$$

are two solutions of the equation $(1+px)^2=x$ on x. By the monotonicity of quadratic polynomials, for all $\alpha\le x<\beta$, $\alpha<(1+px)^2<x$; for all $0<x\le\alpha$, $x\le(1+px)^2\le\alpha$.

Therefore if $\alpha\le a_0<\beta$, for all n, $\alpha<a_{n+1}<a_n<\beta$. Applying the monotone convergence theorem, $\{a_n\}$ has a limit L. Since $L=\lim_{n\to\infty}(1+pa_n)^2=(1+pL)^2$, L must be α (it cannot be β because $a_n<a_0<\beta$).

If $0<a_0\le\alpha$, for all n, $0<a_n\le a_{n+1}<\alpha$. Similarly we deduce that $\{a_n\}$ approaches to α by applying the monotone convergence theorem. □

Take c_d as any node in the $(L-d)$-th level of the tree. Then

$$\mathbb{E}\left[\exp\left(tX_{c_0}\right)\right] = \left(1+\frac{1}{n}(e^t-1)\right)^{nZ} < e^{Z(e^t-1)} \quad (28)$$

Consider the sequence $\{a_n\}_{n\ge0}$ defined by $a_0 = e^{Z(e^t-1)}$, $a_{n+1}=(1+e^{-Zt}a_n)^2$. Since $\mathbb{E}\left[\exp\left(tX_{c_0}\right)\right] < a_0$, applying inequality (25), we deduce by induction that $\mathbb{E}\left[\exp\left(tX_{c_d}\right)\right] < a_d$ for every $d\ge0$.

According to lemma 2, the sequence $\{a_n\}_{n\ge0}$ converges if

$$e^{Z(e^t-1)} < \frac{1}{2}e^{2Zt}(1-2e^{-Zt}+\sqrt{1-4e^{-Zt}}) \quad (29)$$

$$\iff e^{Z(e^t-2t-1)} < \frac{1}{2}(1-2e^{-Zt}+\sqrt{1-4e^{-Zt}}) \quad (30)$$

When $t=1$, the left side of (30) is smaller than 0.6, and the right side is larger than 0.7. For a sufficiently large t, the left side goes to infinity but the right side is smaller than 1. Thus we can denote $\gamma_Z>1$ to be the maximum value of t such that the inequality (30) holds.

For some small $\delta > 0$, let $t = \gamma_Z - \delta$. Now $\{a_n\}_n$ converges to $2(1-2e^{-Zt}+\sqrt{1-4e^{-Zt}})^{-1}$ based on lemma 2. Thus for a sufficiently large L,

$$a_L < \frac{2}{1-2e^{-Zt}+\sqrt{1-4e^{-Zt}}} + \epsilon \tag{31}$$

The stash overflows if and only if the root contains more than $Z+R$ blocks. By Markov's Inequality,

$$\begin{aligned} p_{\text{overflow}} &= \Pr\left[X_{c_L} > Z+R\right] && (32)\\ &= \Pr\left[\exp(tX_{c_L}) \geq e^{t(Z+R+1)}\right] && (33)\\ &\leq e^{-t(Z+R+1)}\mathbb{E}\left[\exp\left(tX_{c_L}\right)\right] && (34)\\ &\leq e^{-(\gamma_Z-\delta)(Z+R+1)}H_Z && (35)\\ &= C_Z\alpha_Z^R && (36)\end{aligned}$$

is the desired upper bound of overflow probability, where

$$\begin{aligned} H_Z &= \left(\frac{2}{1-2e^{-Z(\gamma_Z-\delta)}+\sqrt{1-4e^{-Z(\gamma_Z-\delta)}}} + \epsilon\right) && (37)\\ C_Z &= e^{-(\gamma_Z-\delta)(Z+1)}H_Z && (38)\\ \alpha_Z &= e^{-(\gamma_Z-\delta)} && (39)\end{aligned}$$

In particular, pick $\delta = 0.01$ and a sufficiently small ϵ,

$$\begin{aligned} &C_2 < 0.0407, \quad \alpha_2 < 0.313 \quad (\text{when } N > 2^{12}) && (40)\\ &C_3 < 0.0076, \quad \alpha_3 < 0.291 \quad (\text{when } N > 2^{10}) && (41)\\ &C_4 < 0.0021, \quad \alpha_4 < 0.289 \quad (\text{when } N > 2^{10}) && (42)\\ &C_5 < 0.0006, \quad \alpha_4 < 0.288 \quad (\text{when } N > 2^{10}) && (43)\end{aligned}$$

Appendix C. Correctness of Oblivious Placement

First we formalize the process of Oblivious Placement in Algorithm 4. Without loss of generality, let 0 denote the invalid key. Let $n = 2^k$, and the outmost loop of Oblivious Placement is iterated for k times. In the r-th ($r = 1, 2, \ldots, k$) time, the variable $T = 2^{k-r}$, and we denote it as the r-th round. Let $a^{(r)}$ be the array of the keys (i.e., addresses) at the end of the r-th round. In particular, $a^{(0)}$ is the original array. In the r-th round, for any $M \in \{0, 2T, \ldots, n-2T\}$ and $i \in [M, M+T)$, if $0 < a_{i+T}^{(r-1)} < M+T$ or $a_i^{(r-1)} \geq M+T$, then $a_{i+T}^{(r-1)}$ and $a_i^{(r-1)}$ are swapped in $a^{(r)}$; otherwise they are kept unchanged in $a^{(r)}$.

To prove the correctness of Oblivious Placement, we need to prove that after the final round k, $a_i^{(k)} = i$ for any i appearing in $a^{(0)}$, and $a_i^{(k)} = 0$ for all other i's.

Proof. We will prove that for each $r \in \{1, \ldots, k\}$, and for each $M \in \{0, 2T, \ldots, n-2T\}$ where $T = 2^{k-r}$, the subarray $a_{[M,M+2T)}^{(r)}$ satisfies the following properties:

Property 1. Nonzero elements are in the same range of indices of the subarray. Formally speaking, if the indices of the subarray take in the interval $[p, q)$ (in this case $p = M, q = M+2T$), any nonzero element lies in the interval $x \in [p, q)$.

Property 2. We can rotate the elements such that the indices of all nonzero elements are continuous and strictly increasing. Formally speaking, If the elements in $a^{(r)}$ are not all zero, there exist some $u, v \in [p, q)$ such that either $a_u^{(r)} < a_{u+1}^{(r)} < \cdots < a_v^{(r)}$ (then $u \leq v$), or $a_u^{(r)} < a_{u+1}^{(r)} < \cdots < a_{q-1}^{(r)} < a_p^{(r)} < a_{p+1}^{(r)} < \cdots < a_v^{(r)}$ (then $u > v$); and all other elements in $a^{(r)}$ is 0.

If a subarray satisfies the above properties, say it is (u, v)-cyclic, or cyclic in short, where u, v are taken the same values as they are in property 2.

These properties can be proved by induction. The case when $r = 0$ is just a re-statement of the assumption for the elements in $a^{(0)}$. It suffices to prove the $(r+1)$-th case if smaller cases are proved. Again let $T = 2^{k-r}$. For each $M \in \{0, 2T, \ldots, n-2T\}$, the subarray $B = a_{[M,M+2T)}^{(r)}$ will be separated into two subarrays $C_1 = a_{[M,M+T)}^{(r+1)}$ and $C_2 = a_{[M+T,M+2T)}^{(r+1)}$ after the $(r+1)$-th iteration.

For an index $u \in [M, M+T)$, if $a_u^{(r)} \geq M+T$, say u is raised; for an index $u \in [M+T, M+2T)$, if $0 < a_u^{(r)} < M+T$, say u is sunk. According to the induction hypothesis, B is (u, v)-cyclic for some u, v. We will discuss all possible cases of u and v, and prove that under all cases, both C_1 and C_2 are cyclic.

Case 1. All elements in B are 0. Then all elements in C_1 and C_2 are also 0.

Case 2. $u \leq v$. Because nonzero elements in B are strictly increasing, it is impossible that both sunk and raised indices exist.

If there are neither sunk nor raised elements, then B is unchanged from $a^{(r)}$ to $a^{(r+1)}$. According to the definitions of sunk/raised, property 1 is automatically satisfied for C_1 and C_2. Furthermore the continuous set of indices of nonzero elements, after cutting by half, are still two continuous sets of indices. Hence property 2 is also satisfied.

If there is some index that is raised or sunk, according to the symmetry, we may assume there is a sunk index without loss of generality.

If $u < M+T$, there is some $w \in [M+T, v]$ s.t. $[M+T, w]$ is the set of all sunk indices. Note that if no index is sunk, we set $w = M+T-1$. We will follow this convention when choosing w throughout the proof. Because $a_w^{(r)} < M+T$ (by the definition of sunk), $a_w^{(r)} \geq a_u^{(r)} + w - u$ (elements are strictly increasing), and $a_u^{(r)} \geq M$ (property 1), we have $u > w - T$. Therefore during the $(r+1)$-th iteration, these sunk elements in $[M+T, w]$ are swapped with the elements in $[M, w-T]$ which are before $a_u^{(r)}$ and thus all 0. The resulted C_1 becomes $(u, w-T)$-cyclic, and C_2 becomes $(w+1, v)$-cyclic.

If $u \geq M+T$, then there is some $w \in [u, v]$ s.t. $[u, w]$ is the set of all sunk elements. Therefore after swapping, C_1 is $(u-T, w-T)$-cyclic and C_2 is $(w+1, v)$-cyclic.

Case 3. $u > v$. If there are neither sunk nor raised elements, then B is unchanged from $a^{(r)}$ to $a^{(r+1)}$. Again

property 1 is automatically satisfied for C_1 and C_2. The nonzero elements in C_2 are still continuous, and the nonzero elements in C_1 are either continuous or continuous after rotating. In both cases C_1 and C_2 are both cyclic.

If there exists some sunk or raised element, we will consider three subcases.

1) $a_M^{(r)} < M+T$. Now $a_u^{(r)} < a_M^{(r)} < M+T$, implying $[u, M+2T)$ are all indices being sunk. And there exists some $w \in [M+1, v]$ s.t. $[w, v]$ is the set of all raised indices. Because $a_{w-1}^{(r)} \geq a_u^{(r)} + (w-1+2T) - u$ (elements are strictly increasing), $a_{w-1}^{(r)} \leq M+T-1$ (because it is not raised by the definition of w), and $a_u^{(r)} \geq M$ (property 1), we have $w-1 < u-T$. Thus sunk/raised elements only swap with zeros. Eventually C_1 is $(u-T, w-1)$-cyclic and C_2 is $(w+T, v+T)$-cyclic.
2) $a_M^{(r)} \geq M+T$ and $u \geq M+T$. Then all indices in $[M, v]$ are raised. And there exists some $w \in [u, M+2T)$ s.t. $[u, w]$ is the set of all sunk indices. Because $a_v^{(r)} \geq a_{w+1}^{(r)} + (v+2T) - (w+1)$, $a_{w+1}^{(r)} \geq M+T$, and $a_v^{(r)} < M+2T$, we have $w+1 > v+T$. Thus C_1 is $(u-T, w-T)$-cyclic and C_2 is $(w+1, v+T)$-cyclic.
3) $a_M^{(r)} \geq M+T$ and $u < M+T$. Again all indices in $[M, v]$ are raised. For some $w \in [M+T, M+2T]$, $[M+T, w]$ is the set of all sunk indices. Similar to above, by considering $a_v^{(r)}$ and $a_{w+1}^{(r)}$, we have $w+1 > v+T$. By considering $a_w^{(r)}$ and $a_u^{(r)}$, we have $u > w-T$. Thus C_1 is $(u, w-T)$-cyclic and C_2 is $(w+1, v+T)$-cyclic.

From all cases discussed above, for each r, each one in the corresponding subarrays is cyclic. Particularly, when $r = k$, each subarray is of length 1. According to property 1, the element is either equal to its index, or zero. Because during the rounds the elements are only swapped, every nonzero element in the original array still appears in the result array. Thus our proof is complete. □

Appendix D. Oblivious BFS

Consider using an adjacent list $\mathcal{G}$ to represent the graph. For each vertex v, we denote its row in $\mathcal{G}$ as $(v, c, v_0, v_1, \ldots, v_{c-1})$, where c is the number of the adjacent vertices and v_i is each adjacent vertex. We pre-process $\mathcal{G}$ using Algorithm 5 and use $\mathcal{G}'$ as the input to Algorithm 6. Each item in the output ORAM $\mathcal{V}$ contains its parent vertex, as well as the distance to the source. Note that in the implementation, Line 9 and Line 20 of Algorithm 6 can be merged into one ORAM access, corresponding to path fetch and path eviction, respectively. Considering $\mathcal{V}$ and $\mathcal{G}'$ can be built with BULKOR, the algorithm invokes $4(V+E)+1$ individual ORAM accesses.

Algorithm 5: Oblivious BFS pre-processing

Input: Adjacent list $\mathcal{G}$.
Output: Packed adjacent list $\mathcal{G}'$.

$\mathcal{G}' \leftarrow \varnothing$; $l \leftarrow 0$;
foreach *row* $(v, c, v_0, v_1, \ldots, v_{c-1})$ *in* $\mathcal{G}$ **do**
  Append (l, v, c) to $\mathcal{G}'$, $l \leftarrow l+1$;
  for $i \leftarrow 0$ **to** $c-1$ **do**
    Append $(l, v_i, 0)$ to $\mathcal{G}'$, $l \leftarrow l+1$;
return $\mathcal{G}'$;

Algorithm 6: Oblivious BFS

Input: Packed adjacent list $\mathcal{G}'$, number of vertices V, number of edges E, source vertex s
Output: ORAM $\mathcal{V}$

1 Build an ORAM $\mathcal{L}$ from $\mathcal{G}'$ with BULKOR. For each item (l, v, c_{adj}) in $\mathcal{G}'$, treat l as the logical address and (v, c_{adj}) as the data element, to construct $\mathcal{A}$ and $\mathcal{B}$ in Algorithm 1, respectively;
2 Pick all items (there are V such items) with $c_{\mathsf{adj}} \neq 0$ from $\mathcal{G}'$ using oblivious sort. Build an ORAM $\mathcal{V}$ from these picked items with BULKOR. For each picked item (l, v, c_{adj}), treat v as the logical address and $(l, dis = 0, pa = \perp)$ as the data element;
3 Build an empty ORAM $\mathcal{Q}$;
4 Read $(l_{\mathsf{next}}, dis, pa)$ from $\mathcal{V}$ using address s;
5 $q_{\mathsf{head}} \leftarrow 0$; $q_{\mathsf{tail}} \leftarrow 0$;
6 $c_{\mathsf{rem}} \leftarrow 0$; $dis_{\mathsf{base}} \leftarrow 0$; $pa_{\mathsf{prev}} \leftarrow \perp$;
7 **for** $i \leftarrow 0$ **to** $V+E-1$ **do**
8 Read (v, c_{adj}) from $\mathcal{L}$ using address l_{next};
9 Read (l_h, dis, pa) from $\mathcal{V}$ using address v;
10 $re\_en \leftarrow pa \neq \perp$;
11 $cond \leftarrow c_{\mathsf{adj}} \neq 0$; // whether it is a heading vertex
12 cmov($cond$, l, l_h);
13 cmov($\neg cond$, l, l_{next});
14 cmov($cond$, dis_{base}, dis);
15 cmov($\neg cond \wedge \neg re\_en$, dis, $dis_{\mathsf{base}} + 1$);
16 cmov($cond$, c_{rem}, c_{adj});
17 cmov($\neg cond$, c_{rem}, $c_{\mathsf{rem}} - 1$);
18 cmov($\neg re\_en$, pa, pa_{prev});
19 cmov($cond$, pa_{prev}, v);
20 Write (l_h, dis, pa) to $\mathcal{V}$ using address v;
21 Write l_h to $\mathcal{Q}$ using address $\neg re\_en \cdot q_{\mathsf{tail}} + re\_en \cdot \perp$;
22 cmov($\neg re\_en$, q_{tail}, $q_{\mathsf{tail}} + 1$);
23 $cond \leftarrow c_{\mathsf{rem}} = 0$; // whether all neighbors are visited
24 cmov($\neg cond$, l_{next}, $l+1$);
25 cmov($cond$, q_{head}, $q_{\mathsf{head}} + 1$);
26 Read l_{tmp} from $\mathcal{Q}$ using address q_{head};
27 cmov($cond$, l_{next}, l_{tmp});
28 **return** $\mathcal{V}$;

Appendix E.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

The paper presents a bulk loading algorithm for Path ORAM running in hardware-based trusted execution environment (e.g., Intel SGX). Compared to naively running the ORAM's "write" operation N times or using the only previous specialized algorithm for bulk loading, the new algorithm is a $\log(N)$ factor faster.

E.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

E.3. Reasons for Acceptance

1) An asymptotically faster algorithm with new technical ideas for a potentially important use-case of ORAM.
2) An implementation showing substantial practical improvements over prior approaches.