Towards Practical Oblivious Map

Xinle Cao*
Zhejiang University
xinle@zju.edu.cn

Weiqi Feng*
University of Massachusetts Amherst
weiqifeng@umass.edu

Jian Liu†
Zhejiang University
liujian2411@zju.edu.cn

Jinjin Zhou
Ant Group
zhoujinjin.zjj@antgroup.com

Wenjing Fang
Ant Group
bean.fwj@antgroup.com

Lei Wang
Ant Group
shensi.wl@antgroup.com

Quanqing Xu
OceanBase, Ant Group
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
OceanBase, Ant Group
rizhao.ych@oceanbase.com

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

ABSTRACT

Oblivious map (OMAP) is an important component in encrypted databases, utilized to prevent the server inferring sensitive information about client's encrypted databases based on *access patterns*. Despite its widespread usage and importance, existing OMAP solutions face practical challenges, including the need for a large number of interaction rounds between the client and server, as well as substantial communication bandwidth. For example, the SOTA protocol OMIX++ in VLDB 2024 still requires $O(\log n)$ interaction rounds and $O(\log^2 n)$ communication bandwidth per access, where n denotes the total number of key-value pairs stored. In this work, we introduce more practical and efficient OMAP constructions. Consistent with all prior OMAPs, our constructions also adapt only the *tree-based Oblivious RAM (ORAM)* and *oblivious data structures* (ODS) to achieve OMAP for enhanced practicality. In complexity, our approach needs $O(\log n/\log\log n)+O(\log\lambda)$ interaction rounds and $O(\log^2 n/\log\log n)+O(\log\lambda\log n)$ communication bandwidth per data access where λ is the security parameter. This new complexity results from our two main contributions. First, unlike prior works relying solely on *search trees*, we design a novel framework for OMAP that combines *hash table* with search trees. Second, we propose a more efficient tree-based ORAM named DAORAM, which is of significant independent interest. This new ORAM accelerates our constructions as it supports obliviously accessing hash tables more efficiently. We implement both our proposed constructions and prior methods to experimentally demonstrate that our constructions substantially outperform prior methods in terms of efficiency.

PVLDB Reference Format:
Xinle Cao, Weiqi Feng, Jian Liu, Jinjin Zhou, Wenjing Fang, Lei Wang, Quanqing Xu, Chuanhui Yang, and Kui Ren. Towards Practical Oblivious Map. PVLDB, 18(3): 688 - 701, 2024.
doi:10.14778/3712221.3712235

*These authors contributed equally to this work.
†Jian Liu is the corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712235

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/WeiqiNs/DAORAM.

1 INTRODUCTION

Oblivious algorithms [8, 14, 32, 57, 64] serve as a critical mechanism frequently employed alongside encrypted databases (EDBs) [20, 24, 48] to uphold users' data privacy. They ensure that the access patterns remain independent of the database contents [31]. Therefore, during query processing, an untrusted server gains no information beyond query types, database size, and the size of query results [13]. Recently, there has been a surge in the usage of oblivious algorithms with EDBs [17, 36, 44, 69], which is driven by concerns regarding the security implications of query access pattern leakages [42, 46, 51].

Oblivious map (OMAP) [56, 64] is a specific type of oblivious algorithm designed to facilitate oblivious access to key-value (KV) stores [56], one of the most used database formats in production [67, 68]. OMAP offers clients the security guarantee that an untrusted server, holding the encrypted KV pairs, cannot obtain information regarding which data pair was accessed during query processing, nor its content. Furthermore, OMAP is often used to construct oblivious algorithms for executing more complex queries such as join [17], aggregate [24], and range query [16] in other types of databases. However, designing efficient and practical OMAPs presents a significant challenge. Predominantly, most existing oblivious algorithms rely on the established cryptographic primitive called oblivious RAM (ORAM) [31]. This primitive is a generic tool for achieving obliviousness as it was originally proposed to access *memory* obliviously in random access machine. Specifically, given KV pairs $\{(k_i, v_i)\}_{i=0}^{n-1}$ where keys are consecutive integers (which is used to simulate memory), ORAM supports obliviously accessing one pair from them in functionality. While sharing similarities with ORAM, OMAP is more general and powerful since OMAP supports KV stores, even when the keys are **non-consecutive and arbitrary strings**. This difference incurs a huge gap in their designs such that OMAP cannot be naively constructed from ORAMs with practicality. It raises the following important question:

How can we design an efficient OMAP based on practical ORAMs, requiring only small client-side storage like $O(\log n)$?

Oblivious data structure. Some prior works [13, 24, 56, 64] also attempt to address OMAP through another way, i.e., the use of

Table 1: Comparison of approaches for oblivious map. n is the number of KV pairs stored, λ is the security parameter. β is a *constant* set for branching factor in B/B+ tree [24, 56], thus ObliDB [24] still expresses its interaction and bandwidth as $O(\log n)$ and $O(\log^2 n)$, respectively.

ODS Method	Interaction Round	Communication Bandwidth	Note
ODS+AVL [13, 64]	$O(\log n)$	$O(\log^2 n)$	Many interaction rounds
ODS+B/B+ [17, 24, 56]	$O(\log n/\log \beta)$	$O(\beta \log^2 n/\log \beta)$	Larger bandwidth blowup
Ours	$O(\log n/\log\log n) + O(\log \lambda)$	$O(\log^2 n/\log\log n) + O(\log n \log \lambda)$	Better rounds and bandwidth in practice

oblivious data structure (ODS). In short, ODS refers to oblivious algorithms designed specially for some data structures such as trees and stacks in order to support obliviously accessing these structures *more efficiently* than using the generic ORAM. Wang et al. [64] are the first to define ODS and non-trivially adapt tree-based ORAM to achieve this goal. They introduced an OMAP construction employing ODS for an AVL tree, ensuring that client-side storage does not exceed $O(\log n)$. Since then, this construction has been widely implemented in plenty of works [11, 23, 30] due to its simplicity. The state-of-the-art work on OMAP [13, 62] continues to use this approach as a foundation, incorporating several new optimizations. While the AVL tree makes the OMAP have a good theoretical communication bandwidth, it may not be the optimal choice among search trees in practice, as noted in [56, 64].

Consequently, some works introduce OMAPs based on other types of search trees, including B/B+ trees [17, 24] and a variant similar to B-trees [56], to reduce interaction rounds and improve efficiency. However, these new OMAPs reduce interaction rounds at the expense of increased theoretical communication bandwidth. We summarize the complexity of all existing OMAPs that adapt only the practical tree-based ORAMs in Table 1. The table demonstrates that, to achieve $O(\log n/\log \beta)$ interaction rounds per access, prior works require a larger communication bandwidth of $O(\log^2 n/\log \beta)$, where β is a constant integer predefined by the client. The value of β implies a trade-off between interaction rounds and communication bandwidth. It cannot be too large, as this would result in prohibitively high bandwidth costs. For instance, with $\beta = n$, the communication bandwidth reaches $O(n)$, equivalent to downloading the entire database. Therefore, the value of β must be chosen carefully to adapt to specific applications. Additionally, the communication bandwidth remains still $O(\log^2 n)$ regardless of β, which can be a bottleneck for OMAP when implemented in secure enclaves [62]. Therefore, we ask the following question:

Can we propose new ODS that achieve both fewer interaction rounds and reduced communication bandwidth for more efficient OMAPs?

In this work, we revisit the two questions above and provide a positive answer. Specifically, we propose several new constructions which **are the first to build OMAPs via combining both new novel tree-based ORAMs and ODSs**. These constructions are the first to overcome the $O(\log^2 n)$ communication bandwidth barrier, marking a significant theoretical improvement in OMAP bandwidth [62]. Furthermore, they require only $O(\log n/\log\log n)+ O(\log \lambda)$ interaction rounds per operation. Based on these merits, the proposed methods are far more efficient than prior approaches.

1.1 Overview

Framework. We first introduce a new simple but effective framework for designing more efficient OMAPs. Prior methods organize KV pairs as a search tree and then construct an ODS for the search tree. To improve this approach, we explore the use of hash tables, which are well-known for their efficiency in mapping [21]. However, oblivious hash tables are not ideal in this context due to their expensive costs for achieving obliviousness. As discussed in [64], oblivious hash tales can be achieved via a tree-based ORAM [60] with only $O(\log n)$ client-side storage, but one access to the table requires three accesses to the ORAM for addressing collision in the table. As accessing the tree-based ORAMs with the $O(\log n)$ client-side storage is often costly, i.e., $O(\log n)$ interaction rounds and $O(\log^2 n)$ communication bandwidth per access, oblivious hash tables via ORAMs are considered to have the same complexity and impracticality as OMAPs constructed from ODS for the search tree.

Surprisingly, recent works over the last decade [26, 39] demonstrate that accessing the tree-based ORAMs with limited client-side storage can be done more efficiently. Nevertheless, they still encounter some non-trivial practical problems, which prevent their easy adaption in real-world implementations. In this paper, we will show *how to address these problems* elegantly and propose a much more practical and efficient ORAM protocol called DAORAM (**d**e-**a**mortized ORAM), which is a contribution of substantial independent interest. DAORAM can complete each access with only $O(\log n/\log\log n)$ interaction rounds and $O(\log^2 n/\log\log n)$ bandwidth. With the new advanced ORAM, now we can follow the approach in [64] to naively achieve an oblivious hash table without collision and an OMAP with better complexity. To obtain even more optimized OMAP constructions, we propose a new framework consisting of two components, as outlined below:

- **ORAM for hash table.** We initialize an ORAM to store a hash table of size n *which allows collisions.* For each integer $i \in \{0, \ldots, n-1\}$, we map i to g_i and store this mapping in the ORAM, where g_i is used to record the group of collided KV pairs, i.e., any (k, v) such that $\mathsf{Hash}(k) = i$, where $\mathsf{Hash}(\cdot)$ is a hash function randomly mapping a string to an integer in $\{0, \ldots, n-1\}$.
- **Group OMAP.** The length of g_i is limited and cannot store all KV pairs mapped to i. To address this, we establish an OMAP for all collided pairs at position i. Thus, g_i only needs to store metadata about the OMAP, requiring only $O(\log n)$ bits. There are n distinct OMAPs, as we build an OMAP for each g_i where $i \in \{0, \ldots, n-1\}$. We store them in the same ORAM to prevent the server from observing *which group OMAP* is accessed during query processing.

To summarize, *we handle the collisions in hash tables by utilizing smaller OMAPs for collided KV pairs.* When the client accesses a KV pair (k, v) where $\mathsf{Hash}(k) = j$, it retrieves the corresponding g_j from the hash table ORAM. Then, it uses g_j to find (k, v) from the OMAP storing the collided pairs. The overhead of our OMAP is equal to the sum of the overhead in accessing the ORAM and the group OMAP. Importantly, accessing the group OMAP can be

much more efficient than accessing the ORAM as each group has at most $O(\lambda)$ collided pairs [22, 54]. Hence, our framework allows us to use *only one access to the ORAM and a much cheaper access to the group OMAP* to replace the *three accesses* to the ORAM in the general construction of an oblivious hash table without collision proposed by [64], significantly improving practicality.

Contributions. We summarize our contributions below.

(1) *A new OMAP framework.* We propose a new OMAP framework that combines both ORAM for hash tables and ODS for search trees. Within this framework, we can apply a prior theoretically elegant *tree-based* ORAM scheme [39] and existing OMAPs[13, 24, 56, 64] to present several new OMAP constructions. Compared with prior OMAPs, they are **asymptotically better** and *do not* require any additional expensive techniques.
(2) *A faster ORAM.* We identify the infeasible worst-case performance and impracticality of the ORAM protocol [39] used in our constructions, which makes them unacceptable in production. To this end, we introduce a new **d**e-**a**mortized ORAM protocol named DAORAM. It offers substantially better performance and greater practicality compared to [39], making our constructions indeed outperform all prior OMAPs **both theoretically and practically**.
(3) *Full-fledged Implementation.* We implement three typical prior OMAPs including the widely used baseline [64] and SOTA works [13, 24] and our three new OMAP constructions based on them. We provide a comprehensive evaluation of our DAORAM and OMAPs, demonstrating the significant speedup of our framework to prior OMAPs. The experimental results show that our OMAPs improve **processing time by up to 72.0% and communication bandwidth by up to 92.6%** compared to the SOTA work [13].

2 PRELIMINARIES

In this section, we introduce some basic and important notions used in this work. All notations in this work are introduced as needed, a summary table of notations is provided in [10]. All algorithms are assumed to be probabilistic polynomial-time (PPT).

Pseudorandom function. Following [41], we call a function $F : \{0,1\}^{k_1} \times \{0,1\}^{k_2} \to \{0,1\}^{k_3}$ a *pseudorandom function* (PRF) if:

- There is a polynomial-time algorithm: given a key $K \in \{0,1\}^{k_1}$ and an input $x \in \{0,1\}^{k_2}$, it computes $F_K(x) = F(K, x)$.
- For any PPT adversary $\mathcal{A}$, its advantage

  $$\mathsf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) = |\Pr_{K \leftarrow\$\{0,1\}^{k_1}}[\mathcal{A}^{F_K(\cdot)} = 1] - \Pr[\mathcal{A}^{\$} = 1]|$$

  is negligible in λ, where \$ above denotes the oracle that implements a random function from $\{0,1\}^{k_2}$ to $\{0,1\}^{k_3}$, $\mathcal{A}^{F_K(\cdot)}$ and $\mathcal{A}^{\$}$ denote that the adversary has access to the oracle of function $F_K(\cdot)$ and random function, respectively.

ORAM and OMAP. Oblivious RAM (ORAM) and oblivious map (OMAP) are very similar in definition and functionality. Generally speaking, they allow the client C to store a database $\mathcal{DB} := \{(k_i, v_i)\}_{i=1}^n$ encrypted on the untrusted server $\mathcal{S}$ and then operate each pair of data *obliviously*, i.e., $\mathcal{S}$ cannot infer which pair is operated by C via observing access patterns during operations. However, they are very different in key-value (KV) stores supported: ORAM is originally proposed to access **memory** obliviously. So it always assumes keys in KV store are consecutive integers to simulate memory. OMAP is more general and powerful as it is designed for all KV stores where keys can be arbitrary and non-consecutive strings. To this end, there is a huge gap between existing ORAMs and OMAPs expected in deployment:

- **Feasible but impractical:** Actually, there are indeed some existing ORAMs (e.g., hierarchy ORAMs [4, 52, 53]) which naturally support non-consecutive keys. Nevertheless, they are highly inefficient due to the large constant factors in overhead complexity, even though some of them [4, 6] achieve the $O(\log n)$ optimal theoretical communication bandwidth of ORAM [47].
- **Practical but infeasible:** When we try more practical ORAMs, only tree-based ORAMs [60, 63] demonstrate relative efficiency and are widely used in EDBs [9, 11, 17, 66]. But tree-based ORAMs, when constrained by *limited client-side storage*, typically $O(\log n)$ for practical applications, are capable of supporting only *consecutive* keys as the ORAM functionality requires. They cannot be naively applied to process a KV store with unpredictable and non-consecutive keys, e.g., the database $\mathcal{DB} := \{(\text{Alice}, \text{Boston}), (\text{Bob}, \text{London}), \cdots)\}$.

These limitations above leave building practical OMAPs via ORAM still unsolved. As OMAP is a fundamental primitive for oblivious algorithms and secure EDBs, building practical OMAPs becomes an imperative task in encrypted databases (EDBs).

In algorithms, both ORAM and OMAP consist of two subroutines:

- **Initialization:** $\mathsf{Init}(n, \lambda) \to (\mathrm{st}_C, \mathrm{st}_{\mathcal{S}})$. On input the (estimated) maximal number of pairs in the database n and security parameter λ. C and $\mathcal{S}$ interact with each other to run this subroutine, and produce client state st_C in C and server state $\mathrm{st}_{\mathcal{S}}$ in $\mathcal{S}$.
- **Access:** $\mathsf{Access}(\mathrm{st}_C, \mathrm{st}_{\mathcal{S}}, k, v) \to (\mathrm{st}'_C, \mathrm{st}'_{\mathcal{S}}, v')$. On input the states $(\mathrm{st}_C, \mathrm{st}_{\mathcal{S}})$ and a pair (k, v), C and $\mathcal{S}$ interact with each other to run this subroutine, and produce the updated states $(\mathrm{st}'_C, \mathrm{st}'_{\mathcal{S}})$. If v is $\perp$, then this is an read operation; v' is set to the value stored in $\mathrm{st}_{\mathcal{S}}$ corresponding to k. Otherwise, this is a write operation, and (k, v) is written in $\mathrm{st}'_{\mathcal{S}}$, where $v' = v$.

ODS. Differing from the ORAM and OMAP primitives that aim to operate a single KV pair, oblivious data structure (ODS) [64] tries to design oblivious algorithms specialized to some data structures, e.g., trees [56], heaps [58], and graphs [13]. This enables operating these data structure more efficiently than using the generic ORAM/OMAP [40]. In other words, while ORAM/OMAP is a generic primitive to build various oblivious algorithms, ODS is the specialized data structure for optimizing some important oblivious algorithms in applications, e.g., ObliDB [24] builds oblivious B+ tree to complete range queries obliviously instead of ORAM/OMAP. Additionally, we remark that, although OMAP is conceptually similar to ORAM, compared with extending ORAM to achieve OMAP, most existing works often achieve OMAP via the ODS for search trees [13, 24, 64] to enhance practicality.

3 REVISIT

In this section, we introduce some intuition and specific constructions of prior ORAM/OMAP. They are necessary components this

work is based on. Especially, we revisit them to point out their shortages, explaining why more advanced constructions are needed.

3.1 Prior Recursive ORAM

3.1.1 Basic Intuition. We first provide some background on how tree-based ORAMs work. As illustrated in Figure 1, a KV pair (k, v) is assigned a random *label*, denoted by $pt \in [n]$, indicating the path this pair is on. $\mathcal{S}$ stores a tree where the ciphertext of (k, v, pt) is guaranteed to be on the path from the root node to the pt-th leaf node. The position map in C records the corresponding pt for each key, resulting in a size of $O(n)$. Each time C searches for (k, v), it first retrieves the corresponding pt from the position map using k and then accesses the path indicated by pt. After C retrieves the pair from the path, its label will be replaced by a new random value, denoted by pt'. This pair will be placed in the path of pt'. To achieve this, C can adopt different eviction strategies [18, 60, 63] to balance various trade-offs. If the eviction process fails, the pair is temporarily placed in the stash and will be retried to evict during next ORAM accesses. It has been proven that with some strategies [60, 63], the stash size exceeds $O(\log n)$ with a negligible probability.

While tree-based ORAM looks perfect, the $O(n)$ position map storage makes them impractical. So prior works [59, 61] introduce the *recursion* technique. It uses a series of smaller ORAMs to store the map but requires the **keys must be consecutive integers**, e.g., suppose the position map is $\{(ck_i, pt_i)\}_{i=0}^{n-1}$ where ck denotes consecutive integers, then C can use $\lceil n/2 \rceil$ blocks in another ORAM to store them: the ith block records $\{(ck_{2i}, pt_{2i}), (ck_{2i+1}, pt_{2i+1})\}$. Such an ORAM and block are called PosMap ORAM and block, respectively. To distinguish them from the original ORAM and block holding KV pairs, we call the original ORAM and block as data ORAM and block, respectively. Besides the number 2, the recursion can be deeper with a larger number here. We call this number recursion degree and denote it as X.

In the most common setting [60], the size of both data blocks and PosMap blocks are set to be $O(\log n)$ [26, 60], the path label pt also needs $\log n$ bits to record the corresponding path. In this way, X can be only a constant. The above example shows that we can apply a PosMap ORAM with $\lceil n/X \rceil$ blocks to store the position map for the data ORAM. As X is a constant, this *recursion* process needs to be repeated for $O(\log n)$ times such that C can ultimately use constant storage to access the data ORAM. Unfortunately, the recursion process requires C to sequentially access PosMap ORAMs from small to large and finally access the data ORAM, incurring $O(\log n)$ interaction rounds and $O(\log^2 n)$ communication bandwidth between C and $\mathcal{S}$. Such expensive costs makes recursive ORAM impractical and motivate some works [26, 39] to improve the recursion process.

3.1.2 Review. Here we review some works that try to enlarge X to be $O(\log n / \log\log n)$ in recursive ORAMs to enhance practicality. Fletcher et al. are the first to enlarge X but in an insecure way, which was fixed by Chan et al. [39] later. For ease of understanding, throughout this paper, we treat the recursion process as traversing a complete X-ary tree and here call each pair as a node in the tree. A KV pair in the PosMap ORAM preserving the index (i.e., pt) of X pairs in the next larger ORAM is described as one internal node recording the index of its X children.

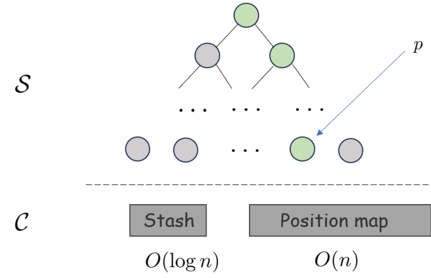

Figure 1: The illustration of tree-based ORAMs.

The main idea of Fletcher et al. [26] is using PRF to generate the index instead of recording the index. In detail, each internal node in [26] consists of three parts: (1) a $\log n$-bit key and $\log n$-bit path; (2) a α-bit group counter (GC); (3) X γ-bit individual counters (ICs):

$$\mathsf{id}||pt||\mathsf{GC}||\mathsf{IC}_0|| \cdots ||\mathsf{IC}_{X-1}.$$

where id is the key of this node and pt is the index (the path where this node is within the ORAM). The values of GC and ICs are initialized as 0. To keep the $O(\log n)$ node size, it is required that $\alpha + \gamma \cdot X \sim O(\log n)$. For this internal node, the recursion process guarantees that the keys of its children are $\{a, a+1, ..., a+X-1\}$ where $a = X \cdot \mathsf{id}$. That's why this node does *not need to store these keys*, leaving the potential to enlarge X. To determine the path of the child with key $a + j$ ($j \in [X]$), C calculates this path based on PRF function, GC, and IC_j. Specifically, C maintains a secret key K for a PRF function PRF and generates:

$$pt_{a+j} := \mathsf{PRF}_{\mathsf{sk}}(a + j||\mathsf{GC}||\mathsf{IC}_j)^1. \quad (1)$$

In the Initialization procedure, C assigns pt_{a+j} to the child with key $a + j$ as its index and this child will be guaranteed to be in the path corresponding to pt_{a+j}. For Access procedure, when C wants to retrieve this child, it gets GC and IC_{a+j} during recursion, calculates pt_{a+j}, and retrieves this path to get this child. After the retrieval, C executes increment:

$$\mathsf{IC}_j := \mathsf{IC}_j + 1 \mod 2^\gamma$$

and reassigns a new path to this child with Equation 1 for eviction placing this child back to the ORAM. In this way, the length of IC can be $o(\log n)$ to allow a larger X. For example, setting $\gamma \sim O(\log\log n)$ and $\alpha \sim O(\log n)$, then they enable $X \sim O(\log n / \log\log n)$.

Security and Fix. There is a vulnerability in the original construction of Fletcher et al. [26]: the value of $\mathsf{GC}||\mathsf{IC}_j$ should not be repeated for any $j \in [X]$ for satisfying computational security. So after C accesses the node with index $a + j$ for $2^\gamma - 1$ times, i.e., the value of IC_j is going to be repeated in the next access towards this node, C is required to change the value of GC and update the path of all the X nodes with the updated GC. This process is called *reset*:

(1) Before updating GC, C retrieves all the nodes with key $\{a, a+1, ..., a+X-1\}$ according to GC and ICs.
(2) C updates $\mathsf{GC} := \mathsf{GC} + 1$, then sets $\forall j \in [X], \mathsf{IC}_j := 0$. Finally, C assigns each node with the new path calculated based on the updated GC and ICs and places them back using eviction.

[1]The level of current node is also taken as an input of PRF, we follow [26] to omit it throughout this paper for ease of presentation.

In the above process, C is required to retrieve and return all X nodes above, making it much expensive. Worse more, the reset happens only when one IC is going to be repeated. As pointed out by Chan et al. [39], now the adversary can infer sensitive information according to the reset frequency. For example, if C is always accessing the same node, then the reset happens very frequently because the same IC is always incremented. However, if C accesses all distinct nodes, no IC is repeated, C will never do reset. So the adversary can infer the pair access distributions according to the reset frequency.

Chan et al. [39] propose a *theoretically* elegant fix where the reset is done randomly. In each access to a child, they do the reset with a probability of $1/X$. So the reset is done independent of the access distribution. In this case, Chan et al. need to guarantee that before any IC is repeated, the reset must have been done to this node. Therefore, they require $\gamma = 3\log\log n$ when X is $\log n/\log\log n$. This promises that repeated GC||IC happens with a probability of $(1-1/X)^{2^\gamma}$ which is negligible in n [39]. Now the cost of reset is still $O(X\log n)$ and the reset is expected to happen once every X accesses. So under this fixed solution, the interaction round and communication bandwidth are $O(\log n/\log X)$ and $O(\log^2 n/\log\log X)$, respectively.

3.1.3 Observations. The fixed approach by Chan et al. [39] is theoretically elegant but leaves some drawbacks in practicality. Here we point out these shortages and *we will address all of them with our new construction in Section 5.*

Observation 1 (Amortized). *The fixed approach guarantees* **only amortized** *interaction rounds of $O(\log n/\log\log n)$ and communication bandwidth of $O(\log^2 n/\log\log n)$.*

This observation is due to the probabilistic reset operations. Suppose the client can store and retrieve at most μ (μ should be a constant) paths once, then the interaction rounds per query are

$$\left\lceil \frac{\log n}{\log X} \right\rceil + \frac{X\cdot u}{\mu}$$

where u is the number of reset operations triggered in the query processing. Also, the communication bandwidth is

$$\log n \cdot \frac{\log n}{\log X} + X\cdot u\cdot \log n.$$

Note u follows the binomial distribution, i.e., $u \sim \mathsf{Bin}(\lceil\frac{\log n}{\log X}\rceil, \frac{1}{X})$. So the query performance actually fluctuates, in the worst case where $u = \lceil\frac{\log n}{\log X}\rceil$, if we assume C can only store one path in local once, the interaction rounds required are as $(X+1)$ times as that in the best case where $u=0$. And obviously, the theoretical complexity in the worst case is also much larger than the amortized complexity. To this end, it is essential to study if we can do deamortization [6, 15, 45, 50] here, i.e., improving the worst-case performance while preserving efficiency. Note *performing stably* is an important property in production [33] and all prior OMAPs and tree-based ORAMs satisfy it, thus without this property, ORAMs and OMAPs may be not competitive to prior works.

Observation 2 (Strict Parameters). *The fix requires γ must be no smaller than $3\log\log n$ and n to be large to guarantee negligible probability $(1-1/X)^{2^\gamma}$.*

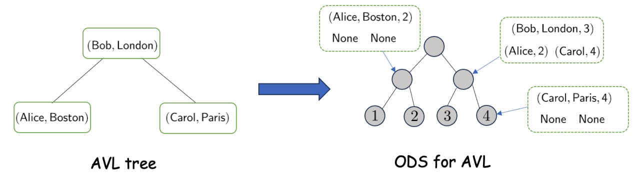

Figure 2: The ODS for AVL. For (Bob, London), the ODS block stores not only the KV pair and its path (Bob, London, 3) but also the children keys and paths (Alice, 2) and (Carol, 4).

These strict parameter values affect the actual performance of the fixed solution. The value of γ implies that if the block size is fixed (like memory blocks), then the upper bound of X is also fixed because we cannot change γ to smaller values than $3\log\log n$. However, with a small X, there can be still too many expensive interaction rounds, making the ORAM inefficient. So we wonder if the value of γ can be smaller for better efficiency. While the smaller values do not imply the improvement on complexity, they are important for actual performance. Besides, another important issue is if we can achieve the security on resets **perfectly**: whatever n is, it is guaranteed that reset must happen before GC||IC in a block is repeated without the sacrifice of obliviousness and efficiency.

3.2 Prior OMAPs

The prior OMAPs [13, 24, 56, 64] organize KV pairs as a search tree according to key orders. To access a pair, C traverses the search tree to find it. Typically, these works apply some classic *data-dependent* search tree such as an AVL tree or a B+ tree. These structures are determined by both database sizes and contents. Traversing the tree requires $O(\log_\beta n)$ interaction rounds where β is the branching degree of a node. Achieving OMAPs naturally involves enabling C to traverse the search tree obliviously, which can be done with a *pointer-based technique* [64] in ODS. In Figure 2, we provide a minimal AVL tree example with data pairs {(Alice, Boston), (Bob, London), (Carol, Paris)}. To preserve the node of an AVL tree, the ORAM tree stores 1) the keys of this node and its children and 2) the paths this node and its children are in. Each time it traverses a node, it finds paths its children located on. Finally, C stores only the root node of the AVL tree.

The ODS of data-dependent search trees can achieve the OMAP with only $O(\log n)$ client-side storage where n is the number of KV pairs in the database. However, they incur in-compressible blocks because the client cannot predict the keys and paths of its children, necessitating these values to be recorded in the block. Recall that the block size in ORAMs is assumed to be $O(\log n)$, given that the key length is also at least $O(\log n)$ [64], each block can store only a constant number of keys, i.e., the branching factor of a node in the search tree can be only a constant. In other words, the design of prior works inherently implies the expensive $O(\log n)$ interaction rounds where the complexity constant factor is determined by the block size and n in production. So up to now, the prior OMAPs [13, 24, 56, 64] cannot overcome *either the $O(\log^2 n)$ communication bandwidth or the $O(\log n)$ interaction rounds while not exceeding $O(\log^2 n)$ communication bandwidth.* To this end, this work tries to design OMAPs under a new novel framework escaping from the above inherent shortages of data-dependent search trees.

4 OMAP FRAMEWORK

In this section, we define the security model for ORAM/OMAP, then we propose our new novel framework for designing OMAPs. This framework allows us to combine the recursive ORAM introduced in Section 3.1 and (modified) ODS for search trees to instantiate new OMAPs which are *asymptotically better* than prior OMAPs.

4.1 Security Model

Consistent with most ORAMs [52, 60, 63] and EDBs [11, 22, 24], we consider a client C that stores its encrypted database (EDB) on a remote, untrusted server $\mathcal{S}$. Typically, C is assumed to have limited storage [26, 39, 60, 64] to accommodate most devices, including those with very limited resources, such as mobile phones, smartwatches, and secure enclaves [62]. The adversary $\mathcal{A}$ is assumed to be *honest-but-curious* adversary $\mathcal{A}$ to capture $\mathcal{S}$. This adversary does not deviate from the predefined protocols or invade the client C, but it observes everything available on $\mathcal{S}$ in the entire process. Specifically, while C issues read and write operations, $\mathcal{A}$ continuously observes the server state to glean as much sensitive information about C as possible.

Definition 4.1 (Security definition). Let $\vec{y_0} := \{(\mathsf{op}_i, ek_i^0, ev_i^0)\}_{i=0}^{m-1}$ and $\vec{y_1} := \{(\mathsf{op}_i, ek_i^1, ev_i^1)\}_{i=0}^{m-1}$ denote two operation sequences with the same length m. The operation type op is either read or write. Let $A(\vec{y_i})$ denote the access sequence of blocks in $\mathcal{S}$ by executing $\vec{y_i}$ via the Access interface of ORAM/OMAP after Initialization[2].

Then an ORAM/OMAP is secure if (1) $A(\vec{y_0})$ and $A(\vec{y_1})$ are computationally indistinguishable by $\mathcal{A}$, i.e., they can be distinguished with an advantage of $\mathsf{negl}(\lambda)$ where λ is the security parameter, and (2) it is *correct*, i.e., the results returned by executing $\vec{y_i}$ via ORAM/OMAP is consistent with that returned by executing $\vec{y_i}$ on unencrypted database directly with a probability of $1 - \mathsf{negl}(\lambda)$, which implies the ORAM/OMAP may fail with probability $\mathsf{negl}(\lambda)$.

The definition of OMAP and ORAM differs only in the KV pairs allowed. While OMAP can process operation sequences with arbitrary keys in the KV store, ORAM assumes all the keys in the KV store can be included by an integer interval with the length set by the initialization (which is because ORAM is simulating the memory). The security definition guarantees that the access patterns do not leak information about the operations besides the operation length. The adversary cannot obtain any knowledge about the operation type or content. Similar to prior works [11, 17, 60], we consider the leakage from side-channel attacks, such as when or how frequently C issues requests, to be out of the scope of this paper. More details and effective defenses to these attacks can be found in [19, 27, 35]. While the initial OMAPs treat both search and insertion as *write* operations and hence, indistinguishable. The recent work [13] allows them to be distinguishable for better search efficiency. Our OMAPs can adaptively allow or disallow these two operations to be distinguished as needed. We discuss this and provide formal security proofs of our OMAPs in the full version [10].

4.2 OMAP Framework

In this section, we propose a new framework for designing OMAPs. Compared with prior OMAPs [13, 24, 56, 64], this framework is the first to combine both tree-based ORAMs and ODS of search trees to achieve OMAP. Moreover, using this framework, we propose new OMAPs with the *best-known* complexity on interaction rounds and communication bandwidths under tree-based structures.

[2]The initialization uses the same value for parameter n which is no smaller than m.

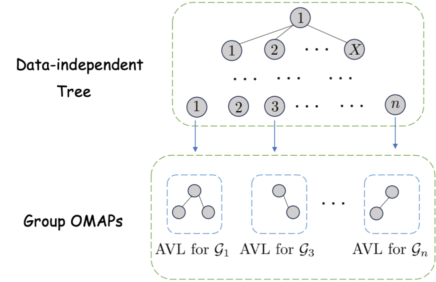

Figure 3: Our constructions under the new framework.

Framework. For better efficiency, we redesign the framework to construct OMAP. Specifically, instead of organizing all the KV pairs as a search tree, we follow the design of the hash table in computer science [21] and recent ORAM works [5, 52]. We divide the data pairs into different groups via a hash function and then adopt efficient methods to access each group and the required pair within it, obliviously. In detail, given a database $\mathcal{DB} := \{k_i, v_i\}_{i=0}^{n-1}$, we perform the following steps:

(1) **Hash:** We randomly map each pair to a group with a hash function $\mathsf{Hash} : \{0,1\}^* \to [n]$:

$$\forall i \in [n], (k_i, v_i) \in \mathcal{G}_j \text{ where } j = \mathsf{Hash}(k_i).$$

It is guaranteed that there are at most $O(\lambda)$ pairs mapped to the same group [29, 59].

(2) **ORAM for consecutive keys:** We apply an ORAM to help access these groups, i.e., we prepare KV pairs $\{(i, pt_i)\}_{i=0}^{n-1}$ where pt_i implies the path to access group $\mathcal{G}_i$. Remark the keys are consecutive so the prior recursive ORAMs [26, 39, 60] (cf. Section 3.1) can be applied here.

(3) **Smaller OMAP for groups:** For each group $\mathcal{G}_i$, we organize pairs within it as a search tree. Then we construct an ODS for the n search trees such that we can access one of them while avoiding C to know which tree is accessed. We call this ODS as *group OMAPs* and it can be achieved by modifying existing OMAPs [13, 24, 64]. As each group has at most $O(\lambda)$ pairs [22, 54], the interaction rounds are reduced to $O(\log \lambda)$. The ORAM in step (2) uses pt_i to record how to access the search tree here for $\mathcal{G}_i$ and then we use it to access pair in $\mathcal{G}_i$.

To search a pair, C first calculates its group $\mathcal{G}_j$ via the hash function, then obliviously accesses $\mathcal{G}_j$ with ORAM to get pt_j, and finally uses p_j to find the pair via the group OMAP for $\mathcal{G}_j$. Under this new framework, we can apply any ORAM in step (2) and any existing OMAP in step (3) to instantiate OMAP construction. Now we explain how this framework enables more practical OMAPs.

Data-independent Tree. We first discuss the ORAM for consecutive keys, especially the recursive ORAMs introduced in Section 3.1. The recursive access in prior ORAMs can be regarded as obliviously traversing a complete X-ary tree from root to leaf where X is the recursion degree, i.e., **the recursive ORAMs actually establish the ODS for a complete X-ary tree**. Interestingly, the complete tree is a *data-independent* tree [25], meaning its structure depends

on only the database size n. So C can exactly predict the next child node accessed when traversing. It does not need to store the keys of children nodes in each traversed node. While it still has to record the paths of children, this information can be compressed because they are only required to be *nearly random instead of specified by application*, as introduced in Section 3.1. Therefore, if we use the same block size, the node of a data-independent tree can include more children than that of the data-dependent tree in prior OMAPs.

Conceptually, our framework needs only an ORAM here for storing data hash information. It is not necessary to use the recursive ORAM and establish the data-independent tree. Applying some more advanced (but impractical) hierarchy ORAMs [4, 52] with optimal complexity, our framework can achieve better complexity than the constructions in this paper, e.g., with the ORAM in [4] and group OMAPs in Section 6, the communication bandwidth can be as low as $O(\log n \log\log n)$ instead of $O(\log^2 n/\log\log n)$ in our constructions. However, we focus on the practicality of OMAPs in realistic scenarios instead of only the theoretical complexity, and up to now, only the recursive tree-based ORAMs [26, 39, 59] have been demonstrated the practicality under $O(\log n)$ client-side storage. To this end, we adopt the recursive ORAM and compress the data-independent tree to improve the ORAM performance.

Group OMAPs. The data-independent tree seems nice but is theoretically equivalent to a simple hash table allowing collisions. There can be a group of KV pairs mapped to the same leaf node in the tree. To this end, after we find the leaf node in the ORAM, we still need to access the required pair within this group via existing OMAPs. Take the OMAP based on ODS+AVL [64] as an example, we establish a new ODS and then organize each non-empty group as an AVL tree stored in this ODS. The height of each AVL tree is $O(\log \lambda)$ as each group has at most $O(\lambda)$ pairs [22, 54]. To access a pair in a group, we traverse only $O(\log \lambda)$ nodes of the AVL tree, and corresponding interaction rounds are also $O(\log \lambda)$.

We still need to clarify how we combine the data-independent tree and group OMAPs. There are two ODSs separately for the two components. Recall we need to know the path of the root node of an AVL tree for traversing the tree. So before we look up the pair required within the ODS for the corresponding group, we first find the path of the root from the ODS for the data-independent tree, i.e., the variable pt_j for group $\mathcal{G}_j$. Until now, we smoothly integrate the two components together for constructing new more efficient OMAPs. We sum the two components for calculation. We conclude the interaction rounds as $O(\log n/\log\log n)+O(\log \lambda)$ and the communication bandwidth as $O(\log^2 n/\log\log n)+O(\log n \log \lambda)$. Note that the ODS for the data-independent tree can be built by the existing recursive ORAM where $X \sim O(\log n/\log\log n)$ [39] and the OMAP for groups can be constructed by any existing OMAP constructions [13, 17, 24, 56, 64]. *So up to now, we can design five OMAP constructions which are asymptotically better than prior works.*

5 DATA-INDEPENDENT TREE

In this section, we propose a new ORAM named **D**e-**a**mortized ORAM (DAORAM). It is motivated by addressing the impracticality of the prior recursive ORAMs [26, 39]. As we pointed out in Section 3.1.3, while the recursive ORAM in [39] can be used to instantiate new OMAPs with better complexity under our framework, it is impractical in production. This makes the OMAPs based on it noncompetitive to prior OMAPs for real-world applications. To this end, we propose DAORAM to address all the shortages of [39] presented in Section 3.1.3 and make our OMAPs indeed practical. For brevity, we still treat the recursion process as traversing a complete X-ary tree and call each KV pair as a node in the tree.

5.1 Construction

In this section, we propose a new recursive ORAM protocol named **D**e-**a**mortized ORAM (DAORAM) for achieving the ODS of the data-independent tree efficiently. Motivated by our observations in Section 3.1.3, there are three design goals for our new protocol:

1. **De-amortization:** It should perform stably, even the worst-case performance is still efficient.
2. **Larger X:** It should enables a large X to reduce interaction rounds as much as possible.
3. **Perfect reset:** There is no repeated GC||IC with *probability 1* whatever the database size is.

Overall, compared with the fixed approach [39], our new protocol is expected to be more practical and efficient.

Reset analysis. Here we explain why the reset operation is expensive and should be "removed". First of all, lots of works [7, 28, 49, 55, 65] have demonstrated that the main cost overhead of ORAMs is *communication* including the interaction rounds and communication bandwidth. That's why a line of work [5, 18, 28, 52, 55] are trying to pursue better interaction rounds and communication bandwidth. The reset operation becomes costly as it needs C to download X paths and then place them back. Although C can retrieve μ paths in parallel (within the same interaction round) to reduce interaction rounds, it needs to provide $O(\mu \log n)$ storage and still interacts with $\mathcal{S}$ for X/μ rounds. Recall we always assume C owns only $O(\log n)$ storage to cover a wide range of devices like smartwatches, which implies μ should be a constant. So the reset cannot avoid $O(X)$ interaction rounds and transferring X paths between C and $\mathcal{S}$. Worse more, C is often assumed to interact with $\mathcal{S}$ **under WAN** [11, 17, 37, 56] where the latency can be high. This makes the multiple interaction rounds in the reset further unacceptably expensive, becoming one bottleneck for real-world applications where low latency is important [28].

Intuition. Now we can introduce the intuition of our construction. The main challenge is how to remove the reset while preserving the efficiency of each query, i.e., each query processing should be as nearly fast as the baseline in the fix, i.e., no reset happens during the query processing. As we have analyzed that the expensive costs in ORAM come from interaction rounds and communication bandwidth, we address them with the following guideline:

1. Removing the $O(X)$ interaction rounds for the reset;
2. Transferring the X paths partially in each query, e.g., transferring only one of the X paths in each query.

In this way, C will feel only a little sacrifice on efficiency because the overhead brought by (2) is very cheap relative to the total time usage. But the practicality is improved much as lots of interaction rounds incurred by resets are not needed any more. To achieve the guideline above, we resort to the de-amortization algorithm to *"spread" the reset operation over many queries.* Specifically, in

this paper, we make use of the interaction rounds in usual query processing to partially transfer reset paths and process the reset.

De-amortization philosophy. De-amortization is a classic topic about ORAM and has been studied a lot [6, 15, 45, 50]. However, all prior works aim to hierarchy ORAMs [4, 53] and cannot be non-trivally applied in tree-based ORAMs. This is because among tree-based ORAMs, only the works above [26, 39] which try to compress children within a node suffer from the worst-case performance. However, such ORAM protocols are important for our framework. As far as we know, this work is the first to introduce de-amortization in tree-based ORAMs and adopt a philosophy different from prior works, which can be helpful for understanding the recursive ORAM and OMAPs based on it.

The prior works [6, 15, 45, 50] hope to prepare a backup for the expensive operation (like the reset) during usual queries. So if C needs reset, it directly starts with the backup and then prepares the next backup. However, this always brings copies of data such that in the end we have to execute de-duplication to delete data copies, which is the main bottleneck in prior works. Moreover, to guarantee data consistency, C has to execute each update query in both the currently used data and backup data which we call as the *current group* and *backup group*, respectively. In this paper, we propose a new lazy strategy for tree-based ORAMs: *pursue the expensive operation instead of preparing it in advance.* In short, if C needs to reset the block when accessing a pair, it just directly **resets this pair alone** and resets all other pairs in this node during the next usual queries. This avoids data copies and also de-duplication, enabling more efficient ORAM construction. The data consistency under our strategy is also guaranteed in a more efficient way. We always preserve each item in only one group. The challenge is to let C always know which group stores the item. Besides, C needs to remove all items from the current data group to the backup group before the next reset is triggered for continuous de-amortization. Our construction proposed below will address the two challenges with practicality.

Data structures. We first define the data structure in an internal node of the data-independent tree. Suppose this node containing the paths of children with key $\{a, a+1, ..., a+X-1\}$, we store two groups of counters: they are the compression for reset and pursuing reset denoted by G^r and G^p:

$$\mathsf{G}^r : \mathsf{GC}^r||\mathsf{IC}_0^r||\mathsf{IC}_1^r||\mathsf{IC}_2^r||\cdots||\mathsf{IC}_{X-1}^r,$$

$$\mathsf{G}^p : \mathsf{GC}^p||\mathsf{IC}_0^p||\mathsf{IC}_1^p||\mathsf{IC}_2^p||\cdots||\mathsf{IC}_{X-1}^p.$$

Consistent to prior works [26, 39], we let GC occupies α bits and IC occupies γ bits such that $\alpha + X \cdot \gamma \sim O(\log n)$. Besides, we **additionally add one bit** $\mathsf{b} \in \{0, 1\}$ as the indicator variable. The path calculation for the child with key $a + j$ ($j \in [X]$) based on the two groups are as below:

$$pt_j^r = \mathsf{PRF}_{\mathsf{K}^{\mathsf{b}}}(a + j||\mathsf{GC}^r||\mathsf{IC}_j^r), \quad (2)$$

$$pt_j^p = \mathsf{PRF}_{\mathsf{K}^{1-\mathsf{b}}}(a + j||\mathsf{GC}^p||\mathsf{IC}_j^p) \quad (3)$$

where $(\mathsf{K}^0, \mathsf{K}^1)$ are two secret keys for PRF. That means we define two different calculations for the two groups and we will show how they are useful in query processing.

Query Processing. Now we describe the specific query processing with three phases as below.

(1) *The initialization phase* happens only once in the beginning. Similar to prior works [26, 39], it initializes $(\mathsf{G}^r, \mathsf{G}^p)$ in an internal node of the data-independent tree to include paths for the children with keys $\{a, a+1, ..., a+X-1\}$. The initial values are set as below:

$$\forall j \in [X], \mathsf{IC}_j^r := 1,\ \mathsf{IC}_j^p := 0.$$

Also $(\mathsf{GC}^r, \mathsf{GC}^p)$ and b are set as 0. The child with keys $a + j$ is guaranteed to be placed in the path calculated by Equation 2.

(2) *The query phase* is for processing a query from C. Here we describe how the access is done between an internal node and its children. C repeats this process for $O(\log n/\log X)$ internal nodes to traverse the data-independent tree. Suppose C wants to access the node with key $a + j$ ($j \in [X]$) in level i, and it has got the internal node in level $i-1$ whose children own keys $\{a, a+1, ..., a+X-1\}$. Then C calculates (pt^r, pt^p) according to Equation 2 and Equation 3. Now C execute procedures according to the value of IC_j^r:
   (a) If $0 < \mathsf{IC}_j^r < 2^\gamma - 1$, C directly retrieves the child node using pt^r. Then C increments $\mathsf{IC}_j^r := \mathsf{IC}_j^r + 1$ to calculate the new assigned path to this child node with Equation 2.
   (b) If $\mathsf{IC}_j^r = 2^\gamma - 1$, i.e., it cannot be incremented more for obliviousness. Next C sets $\mathsf{GC}^p = \mathsf{GC}^p + 1$, $\mathsf{IC}_j^p = 1$. Then C retrieves the child node using pt^r and assigns the node with the new path pt^p to place it back. Finally, C sets $\mathsf{b} = 1 - \mathsf{b}$, $\mathsf{IC}_j^r = 0$, and then swaps (G^r, G^p), i.e., the original reset group now needs to be the one for pursing reset because one value in it reached the upper bound.
   (c) If $\mathsf{IC}_j^r = 0$, then C uses pt_j^p to retrieve the node. Next it increments $\mathsf{IC}_j^r := \mathsf{IC}_j^r + 1$ to calculate the new assigned path to the node with Equation 2. After that, it sets $\mathsf{IC}_j^p = 0$.

   With steps above, C identify the retrieved path and new assigned path so it can write back the child node.

(3) *The reset phase* is executed in parallel with the query phase to reuse the interactions in the query phase. There are two cases:
   (a) If all IC^rs except IC_j^r are non-zeo, then C just retrieves a random path, evicts it and wirtes it back.
   (b) If there exists $j_1 \neq j$ such that $\mathrm{IC}_{j_1}^r = 0$, then C accesses the child node with key $a + j_1$ identically to case (c) in the query phase.

During the execution, every time C issues a query, it interacts with $\mathcal{S}$ to execute the query phase and reset phase in parallel. In total, C will retrieve 2 paths, process them, and return them. The two path are retrieved within the same one interaction round. This guarantees that GC||IC strictly increases if $2^\gamma > X$, the correctness proof is provided in the full version [10].

Remark. The readers may notice that it is not easy for C to always distinguish G^r and G^p correctly because they have the same format and value range. So we will always place G^p behind G^r. The indicator bit b is exactly used to mark which secret key corresponds to the first group.

Reducing groups. Now we have achieved the de-amortization with two compressed groups within an internal node but there

is only one group in prior works [26, 39]. Next, we show how to optimize our construction for reducing group numbers. The setting of two-group parameters helps understand how the reset is partially done per access. But the two groups can be integrated based on a non-trivial observation: for any $j \in [X]$, it holds that one of $(\mathsf{IC}^r_j, \mathsf{IC}^p_j)$ must be zero and the other is non-zero. So we can record only the non-zero value and use only a bit to imply which groups it belongs to. Besides, as GC^r and GC^p can be repeated without sacrificing security, we replaced them with only one variable GC. Now we do the increment GC := GC + 1 every two swaps, i.e., both the logic GC^r and GC^p have been used for recursion. We use the variable b to do this: each time swap happens and also b = 1, we increment GC. Finally, the data structure within a node is:

$$\mathsf{GC}||\mathsf{IC}_0||\mathsf{IC}_1||\cdots||\mathsf{IC}_{X-1} \text{ and } \mathsf{g}_0||\mathsf{g}_1||\cdots||\mathsf{g}_{X-1}||\mathsf{b}$$

where $\mathsf{g}_j \in \{0,1\}$ implies if IC_j belongs to G^r and $\mathsf{b} \in \{0,1\}$.

5.2 Analysis

In this section, we mainly give the analysis of performance to show DAORAM indeed achieves all three design goals in Section 5.1. The proof of correctness and security is formally given in [10]. For the performance, we analyze three metrics including *interaction round*, *interaction bandwidth*, and *computational complexity* per query. In DAORAM, the interaction round is $O(\log n/\log\log n)$ as we still enable $X \sim O(\log n/\log\log n)$ and assume $O(\log n)$ client-side storage. The communication bandwidth is calculated as:

$$O(\log X + \log X^2 + \cdots + \log n) = O(\log^2 n/\log X).$$

where $\log X^i$ denotes the communication bandwidth in the ith interaction round. So when X is $O(\log n/\log\log n)$, the communication bandwidth is $O(\log^2 n/\log\log n)$. To complete the calculation about retrieving a node in ith level, the main computation is sorting the union of $O(\log X^i)$ retrieved nodes and $O(\log X^i)$ nodes in the stash, which requires $O(\log X^i \log\log X^i)$ computation. So the total complexity is

$$O(\log X \log\log X + \log X^2 \log\log X^2 + \cdots + \log n \log\log n)$$

which can be bounded by $O(\log^2 n \log\log n/\log X)$. When X is $O(\log n/\log\log n)$, the total complexity is $O(\log^2 n)$.

It is easy to notice that we successfully remove all interaction rounds for resets by applying the interaction rounds in usual query process. The cost is that we transfer two paths in each access while prior works transfer only one, i.e., we increase one path communication, which is very cheap for the whole query processing. So we achieve the **de-amortization** design goal as expected. Now we explain our de-amortization naturally supports the second and third design goals. For parameters, the correctness and security of DAORAM require only $2^\gamma > X$ even when n whatever n is. This is much more relaxed than that in [39] (cf. Observation 2). Therefore, we can further enlarge X as much as possible to minimize the interaction round for actual performance by solving the following equations to get values of (γ, X):

$$\begin{cases} 2^\gamma - 1 = X \\ \alpha + X \cdot \gamma \sim O(\log n) \end{cases} \tag{4}$$

6 GROUP OMAP

In this section, we introduce the group OMAP under our framework. Its design is creatively modified from existing OMAPs to fit our framework. Under our framework, n KV pairs are randomly divided into n groups. Prior works [29, 59] conclude that each group has at most $O(\lambda)$ items. However, the recent work [22] proposes a theorem that shows this bound can be very low in practice, which makes our method truly more practical than prior OMAPs. For example, given $\lambda = 128$, $n = 2^{24}$, there exists a group consisting of more than 52 pairs with a probability no larger than 2^{-128}. Here we introduce the simplified theorem in [22] here for completeness:

Theorem 6.1. *With n items independently and uniformly randomly mapped to one of n groups, then for the following function $f(n,\lambda)$ that outputs the bound, the probability of there exists one group consisting of more than $f(n,\lambda)$ is negligible in λ.*

$$f(n,\lambda) = min(n, exp[W_0(e^{-1}(\log n + \lambda - 1)) + 1])$$

where $W_0(\cdot)$ is branch 0 of the Lambert W function.

We list a table to identify the small value of $f(n,\lambda)$ under $\lambda = 128$ in [10]. Although we continue using $O(\lambda)$ to bound the group size, the readers should realize this bound can be small enough to be efficient. Now we describe the design and performance of the group OMAP. Recall it is used to store the n groups and access any pair within a group obliviously. For obliviousness, it is required:

- *Group obliviousness:* $\mathcal{S}$ cannot infer which group among the n groups is accessed during the query processing.
- *Pair obliviousness:* $\mathcal{S}$ cannot infer which pair among the $O(\lambda)$ pairs within the group is accessed in the query processing.

OMAPs as a whole. To achieve the pair obliviousness, we can apply any existing OMAP [13, 24, 56, 64] to process the $O(\lambda)$ pairs in the same group. The essence is to guarantee the group obliviousness: *we store all the n groups within the same ODS tree for access.* Take the OMAP based on ODS for the AVL tree as an example, pairs in the same group are organized as an AVL tree. Then the n AVL trees are stored in the same ODS tree. To access a pair in the AVL tree, we just traverse the AVL tree obliviously via the ODS tree. Recall to access the AVL-based OMAP, we need to know how to find the root node of the AVL tree, which is provided by the DAORAM. In the end, the root is updated and rewritten to the DAORAM. As most existing OMAPs [13, 24, 56, 64] are based on a search tree, we can in general apply such a process to all of them for different trade-offs.

Here we introduce three OMAP approaches [13, 24, 64] we use with DAORAM under our framework to instantiate three new specific OMAP constructions. They adopt ODS for different search trees with various trade-offs. We summarize them as below and refer their complexity description to Table 1.

- **ODS+AVL:** This OMAP [64] is based on *oblivious AVL tree.* It is the simplest and achieves the best bandwidth blowup. However, it is the most expensive in reality as pointed out by [56]. Notably, it guarantees the insertion and search are indistinguishable.
- **ODS+AVL*:** This OMAP [13] is also based on *oblivious AVL tree* and is the SOTA work in VLDB 2024. It allows search to be distinguishable from insertion and further optimizes search for better efficiency. It also contributes to the client-side oblivious

algorithm as it is implemented in Intel SGX. As our works focus on the client-server setting [62] where the client-side algorithms are not required for obliviousness, we just adopt more efficient algorithms for this OMAP in $\mathcal{C}$.

- **ODS+B+:** This OMAP [24] is based on *oblivious B+ tree* and thus allows a lower tree height and fewer interaction rounds. However, it achieves the reduced interaction rounds at the cost of communication bandwidth: compared to prior methods [13, 64], the block and bucket size here have to be extended for storing more keys within one node of the B+ tree.

7 EVALUATION

Our proposed new framework for designing OMAP comprises two important components: an ORAM for *the data-independent tree* and an ODS for *group OMAPs*. The new ORAM protocol named DAORAM not only surpasses the performance of prior solutions [26, 39] suited for the data-independent tree, but also achieves *de-amortization* for practicality. The ODS is adapted from existing OMAP schemes described in Section 6. To this end, we combine DAORAM with the three aforementioned OMAPs to build three new OMAP constructions, studying the following two questions:

Q1. What is the performance gain of DAORAM compared to prior ORAMs [26, 39] for the data-independent trees? (Section 7.1)

Q2. What is the performance gain of our new OMAPs compared to prior OMAPs [13, 24, 56, 64]? (Section 7.2)

Settings. Consistent with prior works [13, 24, 56, 64], we implement our ORAM/OMAPs in a client-server setting. $\mathcal{S}$ operates a powerful machine, featuring an Intel Xeon Platinum 8160 CPU (96 cores, 2.10 GHz) and 376 GB memory, located in Hangzhou, China. $\mathcal{C}$ operates on a relatively lightweight Alibaba Cloud machine equipped with 4 vCPUs (from an Intel Xeon Platinum 8269CY, 2.50GHz) and 16 GB memory and located in Beijing, China. Importantly, $\mathcal{C}$ and $\mathcal{S}$ interact with each other over the WAN to simulate reality, with a bandwidth of 100 Mbps and an average latency of 38 ms. Our constructions are implemented in Python 3.10, where the encryption (AES 128-bit) and PRF are imported from the pycryptodome package [3]. To ensure a fair comparison, we implement prior OMAPs following their open-sourced repositories [1, 2] in Python. We follow the commonly used parameters to set up ORAMs: each bucket has 4 blocks, and the block size of DAORAM is set to 512 bits, consistent with [26]. Following prior works [22, 24, 48], we primarily conduct experiments on synthetic datasets, as the obliviousness property guarantees that ORAMs/OMAPs perform independent of data distributions [11, 34]. We generate the synthetic datasets of varying sizes to evaluate the scalability of our proposed constructions.

7.1 Practical ORAM with de-amortization

To demonstrate that DAORAM is the most practical and efficient, we implement DAORAM and two other recursive ORAMs for comparison. We refer to the ORAM constructions by the names listed below in the following discussions:

- Freeset: Freecursive [26] serves as the baseline, although it does not achieve obliviousness. It represents the best possible average processing time for queries, as it requires the fewest resets.

Table 2: Comparison between Fixset and Probset on stability.

Reset number		0	1	2	3
Proset	time (s)	0.29	1.72~2.95	3.14~4.75	4.55~5.06
	band (KB)	24	91.5~392.3	234.2~685.4	452.1~903.3
Fixset	time (s)	**0.37**			
	band (KB)	**39.7**			

- Probset: Freecursive with the probabilistic reset [39] is the only ORAM (before ours) that satisfies both the obliviousness and the data-independent tree requirements. However, it is impractical due to its inefficiency and unstable query performance.
- Fixset: Our DAORAM with the fixed reset not only ensures stable performance but also performs more efficiently than Probset.

De-amortization. We compare Fixset and Probset on their query performance. We run both of them on a synthetic dataset with 2^{24} (over 16,000,000) KV pairs, where both key and value are 4 bytes, and perform 2^{20} queries. Since Probset processes queries with random resets, we categorize its queries based on the number of resets that occur during the query. The corresponding bandwidth and processing time per query are shown in Table 2. The results are intervals when the reset number is non-zero because resets can occur in ORAMs of different sizes, leading to varying costs. Clearly, the costs noticeably increase even if the reset number is only 1. The processing time can be $6 \sim 17\times$ slower than that in the best case, where the reset number is 0. In contrast, Fixset always performs stably, and the results show that its performance is comparable to even the best-case performance of Probset. Additionally, we evaluate the stash size stored in $\mathcal{C}$ to demonstrate DAORAM also outperforms prior works [26, 39] in stash size. We show the maximum stash size used under different query numbers in Figure 4. While the query distribution does not affect the reset operations, it possibly impacts the stash sizes. To study this, we generate queries under two typical query distributions: 1) all queries *repeatedly* access the same pair, and 2) queries follow *uniform* distributions. It is shown that Fixset has an impressively smaller clientside stash than the other two protocols. The stash size in Fixset is only one-third of that in Probset! This results from two advantages of DAORAM brought by de-amortization. Firstly, there are dummy accesses in DAORAM, which only evicts a random path. They help DAORAM reduce the stash size but the other two works do not have such dummy accesses. Secondly, while the other two protocols retrieve only one path per access, DAORAM retrieves two paths per access corresponding the query and reset phase, respectively. This allows more aggressive eviction strategies, reducing the stash size. All these results demonstrate, benefited from de-amortization, Fixset is the most practical among the protocols of interest.

Efficiency. We also compare the average processing time and communication time of all queries to evaluate their performance. With each of the three ORAM protocols, we run 2^{10} queries on databases with sizes ranging from 2^{10} to 2^{24} and present the results in Figure 5. The queries are generated without repetition to ensure Fixset maintains its best-case performance by avoiding any resets. Despite this, Fixset's performance remains close to that of Freeset and improves upon Probset by $21\% \sim 47\%$. Thus, Freeset

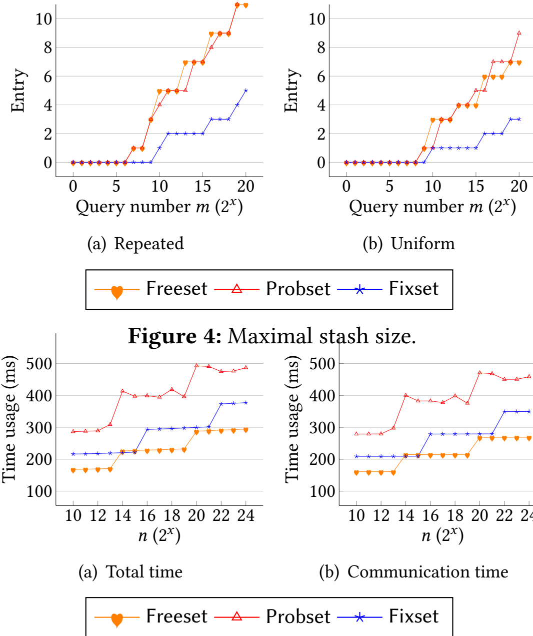

Figure 4: Maximal stash size.

Figure 5: Amortized cost per query.

(i.e.,DAORAM) is highly suitable for production due to its efficiency improvements and practicality.

7.2 Efficient OMAP with less communication

In Section 6, we list three existing OMAP constructions. Under our framework, we construct three new OMAPs using DAORAM with each of these OMAPs. We compare the efficiency of each new OMAP with the corresponding existing OMAP it is based on to demonstrate our framework accelerates OMAPs:

- DAORAM+AVL vs. ODS+AVL: ODS+AVL [64] is the baseline and is included in the comparison as it is the first and most widely-used OMAP [11, 38, 48].
- DAORAM+AVL* vs. ODS+AVL*: ODS+AVL* [13] is the state-of-the-art OMAP based on the AVL tree. It optimizes the search algorithm of the baseline for better efficiency.
- DAORAM+B+ vs. ODS+B+: ODS+B+ [24], benefiting from its large branching degree, is the most efficient OMAP to date.

As claimed in [62] and Section 5.1 of this work, the performance bottleneck of OMAPs in a client/server setting is the bandwidth and interaction rounds. Therefore, we use the following three metrics to evaluate OMAPs: *1) the time usage of operations, 2) the number of interaction rounds, and 3) the communication bandwidth.*

Insertion. We run all six OMAPs mentioned above on databases with sizes ranging from 2^{10} to 2^{24} and execute 100 queries, as in [24, 56]. Here we mainly focus on the insertion operation. Insertion is essential to OMAPs because it captures the write operations on databases and in the original and ideal OMAPs [64], even the search operation should seem identical to insertion. So we present the insertion comparison here and leave the search comparison in [10] for space. The time usage of insertion is depicted in Table 3 and we decompose the time in detail to show the speedup in different components further. Besides, to explain the speedup, we list the improvement in communication of our OMAPs in Table 4.

As shown in the tables, there is a substantial speedup of our OMAPs to prior OMAPs. Firstly, in communication time (T_2 in Table 3), our OMAPs achieve a speedup of 37.0% $\sim$ 72.0% compared to the corresponding OMAPs they are based on. This results from the reduced interaction rounds and bandwidth. We significantly reduce the interaction rounds and bandwidth with a speedup from 35.6% $\sim$ 92.6%! The interaction round is the dominant factor in communication as the OMAPs are run under WAN, hence the speedup of T_2 is closer to that of the interaction round. Also, communication occupies the most time usage during query processing, our OMAP mainly improves communication to enhance efficiency. Secondly, our OMAPs also speed up the client-side calculation (T_1 in Table 3) between 33.3% $\sim$ 71.1% although T_1 owns only a very minor proportion. This speedup comes from reduced bandwidth, which implies we retrieve fewer items to calculate, thus T_1 decreases. The speedup of T_1 is lower than the reduction in bandwidth because the client-side calculation in our OMAPs is more complex, lowering the speedup. Finally, in the whole query processing time (T_3 in Table 3), our OMAPs achieve a speedup of 40.1% $\sim$ 72.0%. The most efficient of our OMAPs is DAORAM+B+, which is almost 6× faster than the baseline ODS+AVL for insertions. We remark our speedup comes from the lower asymptotic complexity, both the two tables show with n increasing, the speedup is more and more significant. So it is predictable the speedup of our OMAPs on insertions will be more pronounced as the database size increases due to the superior complexity of our OMAPs, i.e., our OMAPs are expected to perform even better in very large databases in production.

Extended experiments. There are more extensive experiments conducted to evaluate our ORAM and OMAPs, which are shown in [10]. For DAORAM, we test the impact of different accesses on its stash size, validate its obliviousness by running it under multiple query distributions, and compare it with prior tree-based ORAMs to provide further insights. The results are presented in [10]. For OMAPs, we evaluate more of its operations (e.g., search, delete, etc), validate its obliviousness with six distinct query distributions with different skewness and two different datasets, and finally evaluate it under network conditions across a wide range of latencies. These results are provided in [10].

8 RELATED WORK

The leakage from access patterns has been widely recognized as dangerous in EDBs, prompting handful of works presented in the communities of databases [17, 24, 43], security [48, 56, 64], and cryptography [4, 6, 53] to achieve obliviousness in EDBs. Our work is closely aligned with two well-known areas: *Oblivious RAM* (ORAM) and *oblivious data structure* (ODS).

ORAM. ORAM is an essential primitive that counters attacks based on access pattern leakage, with extensive research in various directions [4, 6, 39, 59, 60]. Our works follow a line of works [12, 26, 39, 55, 63] to improve the relatively practical *tree-based* ORAM. Tree-based ORAMs can be divided into recursive ORAMs [26, 39, 60] and non-recursive ORAMs [63, 66]. Non-recursive ORAMs use $O(n)$ client-side storage to enhance their efficiency, but this storage requirement may be infeasible in production [26, 62]. Recursive ORAMs require small client-side storage, typically $O(\log n)$,

Table 3: Time usage of insertion. T_1 is calculation time, T_2 is the communication time, and T_3 is the total processing time.

n		2^{10}			2^{13}			2^{16}			2^{19}			2^{21}			2^{24}		
Time components		T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
AVL	prior (s)	0.06	2.91	2.97	0.09	3.72	3.81	0.17	4.70	4.87	0.27	5.51	5.78	0.31	6.12	6.43	0.38	6.94	7.32
	ours (s)	0.04	1.25	1.31	0.06	1.48	1.55	0.07	1.58	1.64	0.09	1.81	1.89	0.10	1.82	1.93	0.11	1.94	2.05
	speedup (%)	33.3	57.0	55.9	33.3	60.2	59.3	58.8	66.4	66.3	66.7	67.2	67.3	67.7	70.3	70.0	71.1	72.0	72.0
B+	prior (s)	0.04	1.46	1.57	0.05	1.79	1.84	0.08	2.01	2.09	0.10	2.51	2.61	0.13	2.73	2.86	0.16	2.95	3.11
	ours (s)	0.02	0.92	0.94	0.03	0.94	0.97	0.04	0.98	1.02	0.04	1.00	1.04	0.05	1.04	1.09	0.06	1.08	1.14
	speedup (%)	50.0	37.0	40.1	40.0	47.5	47.3	50.0	51.2	51.2	60.0	60.2	60.2	61.5	61.9	61.9	62.5	63.4	63.3

Table 4: Interaction rounds and communication bandwidth of insertion.

n		2^{10}		2^{13}		2^{16}		2^{19}		2^{21}		2^{24}	
Communicate		round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)
AVL	prior (s)	90	345.6	114	554.49	144	884.74	168	1204.22	186	1476.10	210	1881.60
	ours (s)	58	128.51	58	130.05	60	132.86	60	135.17	60	136.70	62	139.26
	speedup (%)	35.6	62.8	49.1	76.5	58.3	85.0	64.3	88.8	67.7	90.7	70.5	92.6
B+	prior (s)	42	75.26	54	124.42	66	185.86	72	221.18	84	301.06	96	393.22
	ours (s)	28	28.67	28	30.21	30	33.02	30	35.33	30	36.86	32	39.42
	speedup (%)	33.3	61.9	48.1	75.7	54.5	82.2	58.3	84.0	64.3	87.8	66.7	90.0

making them more practical. However, as discussed in Section 2, one data access in recursive ORAMs involves $O(\log n)$ interaction rounds, which can be expensive, especially over WAN. Our work follow [26, 39] to significantly reduce the number of interaction rounds to $O(\log n/\log\log n)$. Notably, our proposed scheme DAO-RAM elegantly avoid the costly worst-case performance in prior works [26, 39]. To our knowledge,DAORAM is the most efficient and practical recursive ORAM protocol to date.

ODS. Since Wang et al. [64] introduced the concept of ODS, extensive research has focused on exploring and improving ODS constructions. Wang et al. [64] propose techniques and constructions for a variety of classic data structures, including trees, sets, and graphs. In particular, they provide the first construction for the oblivious map (OMAP), using an oblivious AVL tree, which is broadly adopted by many EDBs [11, 23, 30] and serves as the baseline for comparisons in this work. Following Wang et al., Roche et al. [56] propose a new tree structure named HIRB (similar to a B tree) to establish a more efficient OMAP. Currently, the SOTA works are [13, 24], which adopt an oblivious B+ tree and an optimized AVL tree, achieving the best performance to date. However, the design philosophy of search trees causes these constructions to be limited by the $O(\log^{1.5} n)$ communication bandwidth lower bound of oblivious search trees, as proven by [40]. Moreover, constructions of search tree based OMAPs have not yet overcome the $O(\log^2 n)$ bandwidth. As far as we know, we are the first to adopt a framework other than the oblivious search tree and achieve $O(\log^2 n/\log\log n)+O(\log\lambda\log n)$ communication bandwidth with OMAP constructions. Our work on ORAM in this paper also suggests that oblivious hash tables [64] can have a similar bandwidth complexity, but as we discussed in Section 1, this approach is still more costly than our constructions. Enigma [62] is another study on OMAP that is independent of our research, as it focuses on optimizing the performance when implementing OMAP in secure enclaves, e.g., the page swaps between inside and outside the enclave. In addition, several other works address OMAP in secure enclaves, with a strong emphasis on achieving obliviousness within the enclave (i.e., the obliviousness in $\mathcal{C}$). All of these works can benefit from our OMAP, as it improves the performance of oblivious algorithms in $\mathcal{S}$. We leave it as future works to integrate our work with prior algorithms in $\mathcal{C}$ to achieve obliviousness practically in both $\mathcal{C}$ and $\mathcal{S}$, a concept referred as *double-obliviousness* in Oblix [48].

9 CONCLUSION

In this paper, we propose a new framework for designing a fundamental oblivious data structure in encrypted databases: *oblivious map* (OMAP). We are the first to combine the *oblivious hash table* (which allows collisions) with an oblivious search tree to build more efficient OMAPs. We propose a new ORAM protocol named DAO-RAM, the most efficient and practical recursive tree-based ORAM so far, for the oblivious hash table. By combining the oblivious hash table with three prior OMAPs based on search trees [13, 24, 64], we present three OMAP constructions and empirically demonstrate that they significantly outperform prior OMAPs. Our work can enhance the efficiency of all encrypted key-value databases and more general encrypted databases.

ACKNOWLEDGMENTS

The work was supported in part by National Natural Science Foundation of China (U20A20222) and National Key Research and Development Program of China (2023YFB2704000). It is also supported by Ant Group. The authors from Ant Group are supported by the Leading Innovative and Entrepreneur Team Introduction Program of HangZhou (Grant No.TD2020001). The authors sincerely thank T-H. Hubert Chan for his helpful comments and suggestions, and also the insightful reviews of VLDB anonymous reviewers especially those on the complexity calculation and security proofs.

REFERENCES

[1] [n.d.]. ObliDB open-sourced repository. ([n. d.]). https://github.com/SabaEskandarian/ObliDB. Accessed in December 2024.

[2] [n.d.]. An open-sourced repository for ODS+AVL. ([n. d.]). https://github.com/obliviousram/oblivious-avl-tree. Accessed in December 2024.

[3] [n.d.]. Python package pycryptodome. ([n. d.]). https://github.com/Legrandin/pycryptodome. Accessed in December 2024.

[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: optimal oblivious RAM. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*. Springer, 403–432.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2022. Optimal Oblivious Parallel RAM. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2459–2521.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. 2023. Oblivious RAM with worst-case logarithmic overhead. *Journal of Cryptology* 36, 2 (2023), 7.

[7] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 837–849.

[8] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 207–218.

[9] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. εpsolute: Efficiently Querying Databases While Providing Differential Privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2262–2276.

[10] Xinle Cao, Weiqi Feng, Jian Liu, Jinjin Zhou, Wenjing Fang, Lei Wang, Quanqing Xu, Chuanhui Yang, and Kui Ren. 2024. Towards Practical Oblivious Map. Cryptology ePrint Archive, Paper 2024/1650. https://eprint.iacr.org/2024/1650

[11] Xinle Cao, Yuhan Li, Dmytro Bogatov, Jian Liu, and Kui Ren. 2023. Secure and Practical Functional Dependency Discovery in Outsourced Databases. *Cryptology ePrint Archive* (2023).

[12] Anrin Chakraborti, Adam J Aviv, Seung Geol Choi, Travis Mayberry, Daniel S Roche, and Radu Sion. 2019. rORAM: Efficient Range ORAM with O (log2 N) Locality.. In *NDSS*.

[13] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proc. VLDB Endow.* 16 (2023), 4324–4338. https://api.semanticscholar.org/CorpusID:265455667

[14] TH Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2018. Cache-oblivious and data-oblivious sorting and applications. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2201–2220.

[15] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2017. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 660–690.

[16] Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian. 2022. Efficient Oblivious Query Processing for Range and kNN Queries. *IEEE Transactions on Knowledge and Data Engineering* 34, 12 (2022), 5741–5754. https://doi.org/10.1109/TKDE.2021.3060757

[17] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 International Conference on Management of Data*. 803–817.

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

[19] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/3052973.3053007

[20] Seung Geol Choi, Dana Dachman-Soled, S Dov Gordon, Linsheng Liu, and Arkady Yerukhimovich. 2021. Compressed oblivious encoding for homomorphically encrypted search. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2277–2291.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[22] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 655–671.

[23] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2019. Dynamic searchable encryption with small client storage. *Cryptology ePrint Archive* (2019).

[24] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017).

[25] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. 2022. Range Search over Encrypted Multi-Attribute Data. *Proc. VLDB Endow.* 16 (2022), 587–600. https://api.semanticscholar.org/CorpusID:252545892

[26] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–116.

[27] Christopher W. Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 213–224. https://doi.org/10.1109/HPCA.2014.6835932

[28] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*. Springer, 563–592.

[29] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13*. Springer, 1–18.

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

[31] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[32] Michael T Goodrich. 2011. Randomized shellsort: A simple data-oblivious sorting algorithm. *Journal of the ACM (JACM)* 58, 6 (2011), 1–26.

[33] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2011. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (Chicago, Illinois, USA) *(CCSW '11)*. Association for Computing Machinery, New York, NY, USA, 95–100. https://doi.org/10.1145/2046660.2046680

[34] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)*. 2451–2468.

[35] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 217–233. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[36] Thang Hoang, Ceyhun D Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. 2018. Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing* 9, 2 (2018), 598–609.

[37] Thang Hoang, Ceyhun D. Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. 2021. Efficient Oblivious Data Structures for Database Services on the Cloud. *IEEE Transactions on Cloud Computing* 9, 2 (2021), 598–609. https://doi.org/10.1109/TCC.2018.2879104

[38] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila Altay Yavuz. 2018. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proceedings on Privacy Enhancing Technologies* 2019 (2018), 172 – 191. https://api.semanticscholar.org/CorpusID:4007767

[39] T-H Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *Theory of Cryptography Conference*. Springer, 72–107.

[40] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. 2019. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2439–2447.

[41] Jonathan Katz and Yehuda Lindell. 2014. Introduction to Modern Cryptography. *(No Title)* (2014).

[42] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340.

[43] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 506–525.

[44] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *arXiv preprint arXiv:2003.09481* (2020).

[45] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM,

143–156.

[46] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 297–314.

[47] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, there is an oblivious RAM lower bound!. In *Annual International Cryptology Conference*. Springer, 523–542.

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

[49] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Paper 2015/668. https://eprint.iacr.org/2015/668

[50] Rafail Ostrovsky and Victor Shoup. 1996. Private Information Storage. Cryptology ePrint Archive, Paper 1996/005. https://eprint.iacr.org/1996/005 https://eprint.iacr.org/1996/005.

[51] Simon Oya and Florian Kerschbaum. 2021. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *30th USENIX security symposium (USENIX Security 21)*. 127–142.

[52] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 871–882.

[53] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer, 502–519.

[54] Martin Raab and Angelika Steger. 1998. "Balls into Bins" - A Simple and Tight Analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98)*. Springer-Verlag, Berlin, Heidelberg, 159–170.

[55] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants count: Practical improvements to oblivious {RAM}. In *24th USENIX Security Symposium (USENIX Security 15)*. 415–430.

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

[57] Sajin Sasy and Olga Ohrimenko. 2019. Oblivious sampling algorithms for private data analysis. *Advances in Neural Information Processing Systems* 32 (2019).

[58] Elaine Shi. 2020. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 842–858.

[59] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *Advances in Cryptology – ASIACRYPT 2011*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–214.

[60] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[61] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[62] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2022. Enigmap : External-Memory Oblivious Map for Secure Enclaves. Cryptology ePrint Archive, Paper 2022/1083. https://eprint.iacr.org/2022/1083 https://eprint.iacr.org/2022/1083.

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

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

[65] Peter Williams and Radu Sion. 2012. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 293–304.

[66] Zhiqiang Wu and Rui Li. 2023. OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption.. In *NDSS*.

[67] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3728–3740. https://doi.org/10.14778/3611540.3611560

[68] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3385–3397. https://doi.org/10.14778/3554821.3554830

[69] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.