# Efficient Mixed-Mode Oblivious RAMs

Wenhao Zhang* Xiao Wang† Chenkai Weng‡

June 5, 2025

**Abstract**

Oblivious RAMs (ORAMs) allow data outsourcing to servers so that the access pattern to the outsourced data is kept private. It is also a crucial building block to enable private RAM access within secure multi-party computation (MPC). In recent years, schemes that match the ORAM lower bound have been proposed in both the outsourcing setting and the RAM-model MPC setting, seemingly putting an epilogue in the theory of ORAM. In this paper, we initiate a study of mixed-mode ORAMs, where accesses to the ORAM are a mix of both public and private accesses. Although existing ORAMs can support public access by treating them as private ones, achieving better efficiency is highly non-trivial.

- We present a mixed-mode ORAM algorithm, assuming the existence of private information retrieval (PIR). When the PIR scheme is communication-efficient, this ORAM achieves the best possible outcome: it has a bandwidth blowup of $O(\log N)$ for private accesses and $O(1)$ for public accesses. This construction can be easily extended for the MPC setting achieving $O(B \log N)$ circuit size for private accesses to $B$-sized blocks and $O(B)$ circuit size for public accesses to the same array.
- We instantiate the above protocol in the three-party computation (3PC) setting with more concrete optimizations, yielding a protocol that performs almost as efficiently as state-of-the-art RAM-3PC protocols for private accesses while being $3\times$ more efficient for public accesses in the LAN setting.

## 1 Introduction

Oblivious RAM (ORAM) allows a client to outsource its encrypted data to a server securely. The security of ORAM ensures that the server cannot learn any information about the data access pattern issued by the client when responding to the client's requests. In recent years, ORAMs have also been found useful in secure multi-party computation (MPC) to achieve sublinear access complexity to arrays [GKK+12]. Generically, this is achieved by having MPC emulate an ORAM client while MPC parties can emulate the ORAM server. In a series of works, it has been shown that the known ORAM lower bounds are tight with matching constructions [GO96, WCS15, LN18, AKL+20, BN16]. This has put an end to the major questions in the research of ORAM. However, to push ORAMs to practice, it is important to study ORAMs with specific features that better suit their application settings. For example, oblivious data structures [ZE13, WNL+14, AHR24] assume certain public correlations between different accesses, e.g., they jointly perform an operation on a data structure; this turns out to allow designing better protocols with reduced cost compared to using ORAM directly. Another example is differentially private ORAMs [WCM18, CCMS19] that relax the indistinguishability requirements when they are allowable in their settings. In this paper, we study yet another setting which commonly happens when ORAMs are used in practice.

In all of the previous ORAM constructions, they all assume that *every* access to the ORAM structure is private. However, in practice, this is not necessarily the case. It is often the case that only a subset

*Northwestern University, wenhao.zhang@northwestern.edu
†Northwestern University, wangxiao@northwestern.edu
‡Arizona State University, Chenkai.Weng@asu.edu

of accesses are private while many other accesses to the same data are public. Examples include ORAM initialization, which can be viewed as performing $N$ number of public write operations; it is often needed to perform batch load that reads data out of the ORAM, which is $N$ number of public read operations; repeated execution of the above two would require even more public access to the ORAM. Another example is statistical analysis on a confidential database where the data is stored in an ORAM. In this scenario, the access pattern for database modification should remain private while the access patterns for analysis algorithms could be public since the algorithm itself is typically not sensitive.

Even in a scenario where only a small portion of accesses are private, it is still infeasible to find a trivial but efficient solution — a natural attempt is to store the (encrypted) data redundantly in both a standard array (for public accesses) and a classical ORAM (for private accesses). However, maintaining consistency still requires treating every public write operation as a private ORAM operation; furthermore, all elements in the array need to be updated and reencrypted after a private write operation. Instead, if we simply deploy a classical ORAM for all operations, a logarithmic overhead will be imposed on every access, even including public accesses, leading to undesirable overhead. In the case of ORAM initialization, the best algorithm [KS14] also has a $O(\log N)$ blowup from oblivious sorting. A natural question is, can we design ORAMs that can efficiently support private and public accesses with better efficiency at the same time?

## 1.1 Our Contributions

In this paper, we initiate the study of mixed-mode ORAMs, which allows both public and private accesses to the same array. We partially answered the above question by showing that, when there is a communication efficient private information retrieval (PIR) scheme, there exist ORAMs that are asymptotically optimal in what is achievable for both the data outsourcing setting and the MPC setting. Finally, we show a concrete instantiation for RAM-based 3PC with mixed-mode accesses.

**Generic construction for optimal mixed-mode ORAM.** First, we show an ORAM construction, with the help of private information retrieval, which can achieve $O(\log N)$ communication blowup for private accesses and $O(1)$ communication blowup for public accesses for any mixed-mode access sequences supporting up to $N$ elements. Inspired by Gordon et al. [GKW18], we adopt a tree-based ORAM but use a PIR scheme to retrieve the path. Because the path is never revealed, it does not need to be refreshed. Then we show that in this case, we can extend this protocol to support public read and write operations while only communicating $O(1)$ data blocks. This scheme requires that the PIR response is "compact", which can be satisfied by most HE-based PIR schemes [MW22, ACLS18, MCR21].

Next, we show that this scheme can be readily supported for the RAM-based MPC protocols, by plugging in a suitable ORAM base scheme, and an MPC counterpart of PIR, where the request and response are all secret shared. In this case, we show RAM-based MPC schemes with circuit size of $O(B \log N)$ per private access and $O(B)$ per public access, where $B$ is the size of the block.

**Concretely efficient mixed-mode ORAM in the 3PC setting.** We apply the above framework to the RAM-model 3PC setting. The state-of-the-art protocol, GigaDORAM [FOSZ23], is highly efficient but does not support efficient public accesses. We adapt our generic framework above to the three-party setting with a customized eviction and access protocol. As a result, our implementation shows comparable performance for private access and $3\times$ faster for public access.

## 1.2 Open Questions

Although our scheme proposes optimal-performing ORAMs, it is not the end but the beginning of this line of research. We summarize some important follow-up questions.

1. As detailed in Section 2.1, we have many evidence to believe that there is a $\Omega(\log N)$ lower bound for mixed-mode ORAM without PIR. However, the classical ORAM lower bound does not appear to be

extendable to this case.

2. From the upper bound perspective, it is not clear if better efficiency of mixed-mode ORAM can be achieved without PIR on $O(N)$ elements.

# 2 Technical Overview

## 2.1 Why Supporting Public Access is Hard?

First, we want to examine the exact challenge in supporting public access in existing ORAM constructions. There are two main paradigms in constructing ORAMs, tree-based ORAM [SCSL11] and hierarchical ORAM [GO96].

**The tree-based ORAMs** put all blocks in a perfect binary tree and assign a dynamic random path for each block. To access a block, the whole path is retrieved and the accessed element needs to be assigned a new random path to break the correlation between this access and the future access to the same element. However, this reassignment has to be done without revealing the new path. Thus the block is put to the root of the tree, and an "eviction" process is invoked to move blocks towards the leaf without revealing each block's destination. The fundamental problem here is that, every time a block is accessed, regardless whether it is a public access or private access, as long as its destination path is revealed, it has to be put back to the root and incurs an eviction cost, which is $O(\log N)$ blowup. As a result, although one could theoretically perform public accesses efficiently, $O(\log N)$-cost is still needed to decouple this public access and other private accesses.

**The hierarchical ORAM** works by maintaining $O(\log N)$ number of oblivious hash tables such that the $i$-th table can hold up to $O(2^i)$ elements. Each element can only appear in $O(1)$ possible locations in each hash table and thus it can only appear in $O(\log N)$ possible locations over the whole structure. When an element is accessed, all possible locations are fetched, and the element is moved to the smallest hash table at the top. Whenever a hash table is full, its elements will be put into the hash table at the next level with new randomness. Thus, for every element, the $O(\log N)$ possible locations are constantly changing after every accesses. Performing public accesses also appears challenging. First, we cannot reveal which level the block is currently located and thus requires reading something from every level; second, if the block is moved to the top level even by public accesses, it needs to be moved down to larger hash tables. During its lifetime, it would contribute to the overall cost of moving elements towards a bigger hash table, with $O(\log N)$ amortized cost.

Spiritually, the two ORAM frameworks share the same philosophy: every element is dynamically mapped to a set of $O(\log N)$ possible locations and this set is refreshed after every access to this element. Therefore, if a public access to this element reveals the set, it has to refresh the possible location set, which would usually cost $O(\log N)$ blow up. Although we could not prove a lower bound, this inherent limitation appears difficult to overcome.

## 2.2 Optimal Mixed-Mode ORAM Scheme

Our first observation is to bring other tools to bypass the above restriction. Specifically, we introduce a private information retrieval (PIR) protocol that allows the client to obliviously fetch an element from a set without revealing the index. By default, PIR is read-only but the server can perform computation as part of the PIR protocol; in contrast, classical ORAM assumes that the server only provides storage service. In prior works, it has been shown that ORAMs with the support of PIR can bypass the well-known ORAM lower bound to achieve $O(\log N/\log\log N)$ blowup in bandwidth cost, when PIR is communication-efficient [AFN+17]. Instead, our focus is to use the power of PIR to equip the ORAM with the ability to efficiently perform public access at any point of the execution.

**Our ORAM scheme.** We first recall the basic structure of tree-based ORAM. All blocks are organized in a perfect binary tree with $O(\log N)$ levels, each node can hold up to $Z$ blocks, and each block contains the index, the payload, and the destination path of the block. The ORAM needs to maintain the invariant that every block is always on the designated destination path stored in the block.

From here, instead of following the typical ORAM structure, we assume that the mapping from element to path is fixed and decided by a public random function $f(\cdot)$. For access to index $i$, the client computes $f(i)$ and uses PIR to retrieve the path without revealing the path. Because the path is not revealed, there is no need to update the path, or maintain a position map to be outsourced recursively. Gordon et al. [GKW18] have shown that in such a setting, the path ORAM eviction algorithm still works with the desired bucket size, and thus private accesses still incur a cost of $O(\log N)$ blocks. [1] Now we look at public accesses. For public read at index $i$, both the client and the server know that the block is on the path $f(i)$ but not where. Therefore, we adopt the optimization similar to the one in Ring ORAM [RFK+15]: the client first downloads the metadata to decide the exact location of the block on the path and then it can run a smaller PIR to retrieve the block from the path without downloading the whole path.

**Supporting public accesses efficiently.** Public write operations are more challenging: if we write the block to the root, it would effectively trigger an extra eviction with cost $O(\log N)$. Although the path is public, if we put the block at a certain level $\ell$, it essentially reveals the fact that there was an empty slot at level $\ell$ before the eviction, which reveals the block distribution on the path thus revealing some information about the prior private access patterns. To overcome this issue, we assign an additional bucket with capacity $O(\log N)$ to each leaf. This bucket is specifically used to hold public-write elements to this path. To maintain correctness, this bucket will be read along with other buckets but no block will be evicted to this bucket. To efficiently write to the bucket, the client can either first download the metadata to know which block is dummy, or the client can build a public mapping to assign the index for every block. Naïvely expanding the bucket would incur an $O(\log N)$ blowup in server storage, but one can "move up" this special bucket to be at $O(\log \log N)$ above the leaf; each bucket is in charge of public write to $O(\log N)$ leaves and one can easily show that the bucket size is still $O(\log N)$ based on the standard balls-and-bins analysis. Finally, one needs to mark all other blocks with the same index as the dummy, which again can be done by metadata manipulation without downloading any block.

In summary, private accesses require using PIR to read $O(\log N)$ blocks from $O(N)$ blocks and sending $O(\log N)$ blocks; private read/write requires using PIR to read 1 block from $O(\log N)$ blocks and send $O(1)$ extra blocks. Therefore, as long as PIR is communication efficient in that the communication is bounded by the block size, we have $O(\log N)$ blowup for private accesses and $O(1)$ blowup for public accesses. Such PIR exists in the single server settings based on homomorphic encryption and is becoming extremely fast recently [MW22, ACLS18, MCR21]; in the two-server setting, it can be constructed based on distributed point functions [GI14].

**Extending to the MPC setting.** The above client-server ORAM can be easily converted into a protocol for MPC in the RAM model, where multiple parties emulate an MPC machine and a virtual client who obliviously accesses its internal memory. Assume the RAM program is fixed before the protocol execution, we can guarantee the access pattern is independent of the protocol. Therefore, the security against a semi-honest adversary is straightforward. To efficiently extend it to the secure computation setting, one needs an MPC-friendly eviction algorithm [WCS15], and also an MPC version of PIR, where the request is computed from an index in secret shares (while the request can be revealed to all parties), and the response needs to be in secret shares as well. It is particularly efficient when instantiated using DPFs [GI14].

[1] Their analysis assumes that the access pattern does not depend on the randomness of the protocol, i.e., non-adaptive ORAM; this holds in the classical ORAM setting where the client is honest and thus also holds when applied for RAM-based semi-honest MPC.

### 2.3 Concretely Efficient RAM-based 3PC

Finally, we instantiate the mixed-mode protocol with concrete efficiency in the three-party setting with an honest majority. Even though there are many results to instantiate the above functionalities with optimal theoretic bounds, it is still highly non-trivial to realize them with a concrete efficiency. We realize it by a suite of customized protocols and general-purpose 3PC based on the replicated secret sharing and garbled circuits. Several optimizations are proposed to reduce the end-to-end running time in both LAN and WAN settings.

Our first optimization is a parallel eviction protocol that flushes the stash and pushes blocks toward leaf buckets. Rather than directly apply an existing eviction algorithm like Circuit ORAM, we carefully design a new eviction algorithm that matches the asymptotic bounds while reducing the running time, by using a large eviction frequency and a paralleled eviction process, inspired by Onion ORAM [DvDF+16, CCR19]. This helps our RAM-based 3PC protocol more efficiently and achieves more than an order of magnitude improvement compared to the protocol that directly apply Circuit ORAM eviction algorithm [JW18].

Our second optimization is a three-party constant-overhead PIR that allows both public and private access to an array in the replicated secret sharing form. Although we can take the existing HE-based PIRs to achieve the same asymptotic efficiency, we explore an alternative design for better concrete performance. Namely, we develop the PIR based on the distributed point function (DPF) [BGI16, BGI15, GYW+23] for three parties, and further adapt it into a private path retrieval (PPR) scheme inspired by techniques from Gordon et al. [GKW18]. Our design leverages the property of replicated secret sharing and results in a light reliance on general-purpose MPC.

We evaluate the performance of this optimized RAM-based 3PC protocol. From the experimental results, we observe that our protocol for public accesses outperforms at least $3\times$ and $2\times$ to the state-of-the-art protocol in the LAN and WAN settings, respectively. Meanwhile, for private accesses it is comparable to those protocols. Our protocol shows a great advantage when adapting it to real-world applications.

## 3 Preliminaries

### 3.1 Notation and Security Model

We use $\lambda$ and $\rho$ to denote the computational and statistical security parameters respectively. We denote by $\log(\cdot)$ the logarithm in base 2. We write $x \leftarrow S$ to denote sampling $x$ uniformly at random from a set $S$. For intergers $a, b$, we define $[a, b) := \{a, \ldots, b-1\}$, $[a, b] := \{a, \ldots, b\}$ and $[a] := [0, a)$. We use bold lower-case letters like $\boldsymbol{x}$ to denote a vector and $x^{(i)}$ to denote the $i$-th component of $\boldsymbol{x}$ with $x^{(0)}$ the first component. Depending on the context, we use $\{0,1\}$ and $\mathbb{F}_2$ interchangeably. Addition in $\mathbb{F}_2$ corresponds to an XOR operation in $\{0,1\}$. We use $\mathbf{unit}(N, \alpha) \in \{0,1\}^N$ for a vector with exact one non-zero entry 1 at position $\alpha \in [0, N)$. For a length-$N$ vector $\boldsymbol{x}$ and $d \in [0, N)$, we define $\mathsf{RightRotate}(\boldsymbol{x}, d)$ as the rotation of $\boldsymbol{x}$ toward right by $d$ positions in a cyclic way. We use $[\![x]\!]$ to denote a secret shared value of $x$.

We use the standard ideal/real paradigm [Can00, Gol04] to prove security of our multi-party protocols in the presence of a *semi-honest, non-adaptive* adversary. In particular, this means that the adversary does not choose their input adaptively based on the randomness of the protocol. As a result, a non-adaptive ORAM is sufficient for the construction.

### 3.2 Arithmetic Black-Box Model

We use the arithmetic black-box model (ABB) to conduct general-purpose secure multi-party computations [DN03]. It is a functionality $\mathcal{F}_{\mathsf{ABB}}$ (Figure 1) that models an MPC machine such that the (input) and (output) interfaces of MPC are interpreted as the storage and retrieval of secret values to and from the

**Functionality $\mathcal{F}_{\mathsf{ABB}}$**

The functionality interacts with $m$ parties $P_0, \cdots, P_{m-1}$.

**Input:** On receiving $(\mathsf{input}, \mathsf{id}, i, x)$ from the party $P_i$ and $(\mathsf{Input}, \mathsf{id}, i)$ from other parties, store $(\mathsf{id}, x)$ in memory.

**Compute:** On receiving $(\mathsf{compute}, \mathsf{id}, \{\mathsf{id}_j\}_{j\in[\ell]}, \boldsymbol{y}, C)$ from all parties, where $\boldsymbol{y}$ are public values and $C$ is a circuit, retrieve $\{(\mathsf{id}_j, x_j)\}_{j\in[\ell]}$ from memory and compute $z := C(\{x_j\}_{j\in[\ell]}, \boldsymbol{y})$. Store $(\mathsf{id}, z)$ in memory.

**Output:** On receiving $(\mathsf{output}, \mathsf{id}, i)$ from all parties, retrieve $(\mathsf{id}, x)$ from memory and send $x$ to $P_i$.

Figure 1: The arithmetic black-box functionality.

machine. The $(\mathsf{compute})$ interface executes a generic circuit computation on any stored values and public inputs. The execution of ABB is reactive so that it can take new inputs and reveal secrets at any step during the computation. For convenience, in the following content we use the notation $[\![x]\!]$ to denote the storage of a secret value $x$ on each party and omit the corresponding id of the secret value.

In instantiating $\mathcal{F}_{\mathsf{ABB}}$ with three parties against a semi-honest adversary who corrupts at most one party, we build upon the framework based on replicated secret sharing [CDI05] and a hybrid of two underlying MPC protocols. Specifically, we use the optimized $(3, 1)$-replicated MPC protocol [AFL+16], and three-party garbled circuits protocol for general-purpose MPC computation [MR18]. The former is optimized for concrete computation and communication overhead, and the latter is optimized for round efficiency.

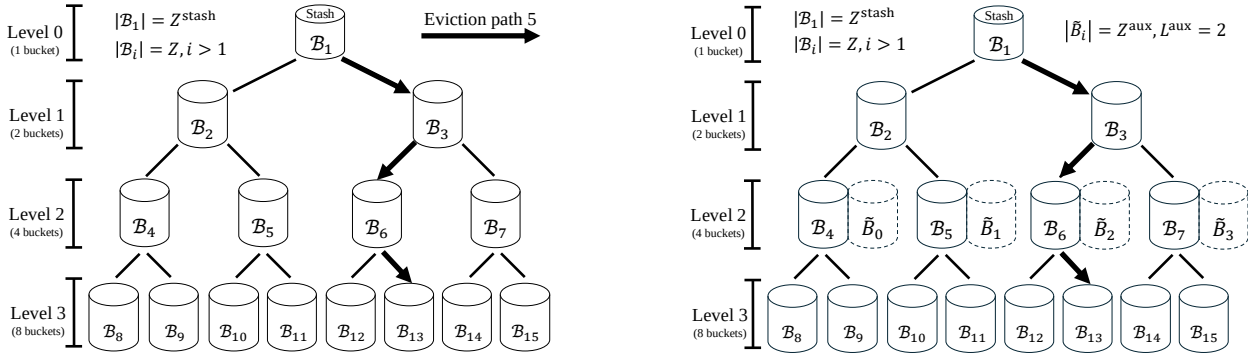
### 3.3 Tree-based ORAM Schemes

The tree-based ORAMs [SCSL11, LLG24, SvDS+13, WCS15, GKW18] are proposed as an alternative approach to the earlier ORAMs based on hierarchical hash tables [Ost90, Gol87]. In tree-based ORAMs, a data structure is constructed in the form of a depth-$L$ perfect binary tree $T$ and each of its nodes is a bucket that can hold up to $Z$ data blocks of the form $(\mathsf{index}\|\mathsf{path}\|\mathsf{data})$. An illustration can be found in Figure 2a.

Denote the level-0 bucket as the root and the level-$L$ buckets as leaves. The tree can store up to $O(2^L)$ data blocks. From the root bucket to the last leaf bucket, the buckets are denoted $(\mathcal{B}_1, \ldots, \mathcal{B}_{2^{L+1}-1})$. The tree is encrypted or secret-shared by one or more servers and allows access by either an external client or by these servers.

Each data block with a logical address $\mathsf{index} = i$ is assigned a random path $r \in [2^L]$ and may reside at any bucket on the path $r$. The most common way to *memorize* the path of each address is by maintaining a position map stored by the client, which records all logical addresses with the corresponding path. This method requires dynamic assignment of paths because each address needs to be randomly re-assigned to another path after each access to achieve obliviousness. Another way is using a keyed pseudorandom function (PRF) with $r := \mathsf{PRF}_k(i) \in [2^L]$ to deterministically bind an address with a path. To prevent leaking access patterns, it requires that the whole path be obliviously retrieved when an address is accessed. Regarding the writing operation, it is much harder to write data directly to its targeted path. Instead, these tree-based ORAMs usually have a temporary buffer called stash. All new data written to ORAM will be stored in the stash and will be evicted to the tree after $A$ operations.

A popular framework for tree-based ORAM schemes is to set the root bucket $\mathcal{B}_1$ as the stash. All newly written data blocks will be temporarily stored at $\mathcal{B}_1$, and will be evicted along their paths afterward. Denote its capacity as $Z^{\mathsf{stash}}$ and the eviction frequency as $A$, a typical eviction strategy is to choose a path $r$ from the stash to the corresponding leaf bucket, and push blocks in buckets along path $r$ to their children




(a) Tree-based ORAM data structure.

(b) Our mixed-mode ORAM data structure.

Figure 2: An illustration of classical tree-based ORAM and our mixed-mode ORAM.

**Functionality $\mathcal{F}_{\mathsf{DowPIR}}$**

This functionality interacts with a client $\mathcal{C}$ and $m$ servers, making a total of $m+1$ parties.

**Initialize:** On receiving $(\mathsf{init}, D)$ from $\mathcal{C}$ and $(\mathsf{init})$ from servers, where $D$ is an array of $N$ elements and each $D[i] \in \{0,1\}^B$ for $i \in [N]$, store $D$ internally.

**Public read:** On receiving $(\mathsf{PublicRead}, i)$ from all parties, output $D[i]$ to $\mathcal{C}$.

**Public write:** On receiving $(\mathsf{PublicWrite}, i, d)$ from $\mathcal{C}$ and $(\mathsf{PublicWrite}, i)$ from servers, where $d \in \{0,1\}^B$, update $D[i] := d$.

**Private read:** On receiving $(\mathsf{PrivateRead}, i)$ from $\mathcal{C}$ and $(\mathsf{PrivateRead})$ from servers, output $D[i]$ to $\mathcal{C}$.

Figure 3: Functionality of PIR for data outsourcing.

buckets [DvDF+16, CCR19]. The eviction path is chosen randomly or in a deterministic way of the reverse lexicographical order to reach an optimal failure probability.

### 3.4 Distributed Point Function

A distributed point function (DPF) is a function secret sharing scheme that secretly shares a point function with two parties. In this paper, we discuss the case that the point function ranging to a boolean value as $f_\alpha : [0, N) \to \{0, 1\}$, where $N$ is a power of 2, $\alpha \in [0, N)$. $f_\alpha(x)$ outputs 1 if $x = \alpha$, and 0 otherwise. A DPF scheme $\mathsf{DPF} := (\mathsf{Gen}, \mathsf{EvalAll})$ works as:

- $(k_0, k_1) := \mathsf{DPF.Gen}(1^\kappa, \alpha)$: outputs a pair of keys $(k_0, k_1)$. We omit $1^\kappa$ in the remaining texts.
- $\boldsymbol{t}_b := \mathsf{DPF.EvalAll}(k_b)$ for $b \in \{0, 1\}$: evaluates the DPF function on the full domain and computes an XOR secret share $\boldsymbol{t}_b$, such that $f_\alpha(x) = t_0^{(x)} \oplus t_1^{(x)}$ for all $x \in [0, N)$.

Existing DPF schemes [GI14, BGI15, GYW+23] can achieve a key size of $O(\log N \cdot \kappa)$ with $O(N)$ communication complexity for full domain evaluation.

## 4 Mixed-Mode ORAMs

In this section, we present our mixed-mode ORAM protocols with the aid of a private information retrieval functionality. We first define the data structure and its parameters, then describe our ORAM scheme in the

client-server model. Finally, we extend the client-server ORAM to secure computation.

### 4.1 Data Structure

For our mixed-mode ORAMs, we design an ORAM data structure $T$ that stores a logical memory $D$ which is an array of $N$ elements, each in the form of $D[i] \in \{0,1\}^B$ for $i \in [N]$. Its architecture is illustrated in Figure 2b. Using the same notations as in the description of tree-based ORAMs in Section 3.3, we define the depth $L$, buckets $(\mathcal{B}_1, \ldots, \mathcal{B}_{2^{L+1}-1})$, and path $r$ for any $r \in [2^L]$. Each bucket contains a number of blocks. We denote $Z^{\mathsf{stash}}$ as the size of the root bucket $\mathcal{B}_1$ and $Z$ as the size of the remaining buckets. Each block is defined as a tuple with four fixed-length fields $(\mathsf{valid}, \mathsf{index}, \mathsf{path}, \mathsf{data}) \in \{0,1\} \times \{0,1\}^{\lceil \log N \rceil} \times \{0,1\}^L \times \{0,1\}^B$. With the initialization of the PRF key $k \in \{0,1\}^\lambda$, if a block is a *real* block, it contains $\mathsf{valid} = 1$, satisfies $\mathsf{path} := \mathsf{PRF}_k(\mathsf{index})$, and resides on the path corresponding to its field $\mathsf{path}$. If a block has $\mathsf{valid} = 0$, then it is a dummy block and its fields $(\mathsf{index}, \mathsf{path}, \mathsf{data})$ may take any arbitrary values.

The main differences between our ORAM and conventional tree-based ORAM schemes are as follows.

- Most of them rely on dynamic position maps while our protocol uses a fixed position map. Each logical address is deterministically mapped to a path of the tree in $T$ using a pseudorandom function $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^{\lceil \log N \rceil} \to \{0,1\}^L$. For a fixed PRF key $k \in \{0,1\}^\lambda$ that is sampled during initialization, the PRF is written as $\mathsf{PRF}_k$ for simplicity.
- Define parameters $L^{\mathsf{aux}} \leq L$ and $Z^{\mathsf{aux}}$, our data structure $T$ stores $2^{L^{\mathsf{aux}}}$ additional buckets $(\tilde{\mathcal{B}}_0, \ldots, \tilde{\mathcal{B}}_{2^{L^{\mathsf{aux}}}-1})$ each of size $Z^{\mathsf{aux}}$. For $i \in [0, 2^{L^{\mathsf{aux}}})$, each of $\tilde{\mathcal{B}}_i$ is attached to the level-$L^{\mathsf{aux}}$ bucket $\mathcal{B}_{2^{L^{\mathsf{aux}}}+i-1}$. They are used to optimize public access. Figure 2b demonstrates an example of these additional buckets in the case of $L^{\mathsf{aux}} = 2$. When instantiating the protocols, the parameters $(L^{\mathsf{aux}}, Z^{\mathsf{aux}})$ will be properly selected to avoid overflow while achieving high efficiency.

**Invariant.** For a logical address $i \in [N]$, let $r := \mathsf{PRF}_k(i)$ be its path index. For $j \in [0, L]$, denote an index $r_j := 2^j + \lfloor r/2^{L-j} \rfloor$ such that $\mathcal{B}_{r_j}$ is the bucket with level $j$ on the path $r$. Additionally, denote $r_{L+1} := \lfloor r/2^{L-L^{\mathsf{aux}}} \rfloor$ such that $\tilde{\mathcal{B}}_{r_{L+1}}$ is the additional bucket attached on the layer-$L^{\mathsf{aux}}$ of the path $r$. Given the feature of our data structure, the mixed-mode protocol maintains the following invariant regarding the buckets $(\mathcal{B}_{r_0}, \ldots, \mathcal{B}_{r_L}, \tilde{\mathcal{B}}_{r_{L+1}})$:

- In each of the above buckets, there exists at most one real block with $\mathsf{index} = i$.
- Among all buckets that contain a real block with $\mathsf{index} = i$, the one resides in $\mathcal{B}_{r_j}$ (or $\tilde{\mathcal{B}}_{r_j}$) with smallest $j \in [0, L+1]$ stores the correct value of $D[i]$.

### 4.2 Client-Server ORAM

We follow the standard client-server ORAM definitions and formalize the functionality that supports both private and public accesses as $\mathcal{F}_{\mathsf{CS\text{-}ORAM}}$ in Figure 4. The functionality interacts with a client $\mathcal{C}$ and $m$ servers. It is securely realized by protocol $\Pi_{\mathsf{CS\text{-}ORAM}}$ shown in Figure 5, which we will elaborate in the following.

**PIR for Data Outsourcing.** The protocol is built upon a data outsourcing with private information retrieval functionality $\mathcal{F}_{\mathsf{DowPIR}}$ (Figure 3). $\mathcal{F}_{\mathsf{DowPIR}}$ allows the client $\mathcal{C}$ to outsource memory to servers, then privately retrieve an element without revealing the index to servers, or publicly update an element on a position known by all servers. It can be instantiated by some previous stateless PIR schemes that are secure against semi-honest adversaries [MW22, ACLS18, MCR21]. We require that these PIRs only incur constant storage on the client and $O(1)$ bandwidth blowup on private retrieval and public updating.

**Initialization.** The initialization phase sets up the publicly-known PRF key $k$ and outsources the memory $D$. Specifically, $\mathcal{C}$ first samples a random key $k$, then locally initializes the data structure $T$ following the invariant: For each $i \in [N]$, $\mathcal{C}$ appends a real block with $\mathsf{index} = i$ and $\mathsf{data} = D[i]$ to the additional bucket

**Functionality $\mathcal{F}_{\mathsf{CS\text{-}ORAM}}$**

This functionality interacts with a client $\mathcal{C}$ and $m$ servers, making a total of $m+1$ parties.

**Initialize:** On $(\mathsf{init}, D)$ from $\mathcal{C}$ and $(\mathsf{init})$ from servers, where $D$ is an array of $N$ elements and each $D[i] \in \{0,1\}^B$ for $i \in [N]$, store $D$ internally.

**Public read:** On $(\mathsf{PublicRead}, i)$, output $D[i]$ to $\mathcal{C}$.

**Public read then write:** On $(\mathsf{PublicReadWrite}, i, g)$ from $\mathcal{C}$ and $(\mathsf{PublicReadWrite}, i)$ from servers, where $g : \{0,1\}^B \to \{0,1\}^B$, output $D[i]$ to $\mathcal{C}$ then update $D[i] := g(D[i])$ internally.

**Private read:** On $(\mathsf{PrivateRead}, i)$ from the client $\mathcal{C}$ and $(\mathsf{PrivateRead})$ from servers, output $D[i]$ to $\mathcal{C}$.

**Private read then write:** On $(\mathsf{PrivateReadWrite}, i, g)$ from $\mathcal{C}$ and $(\mathsf{PrivateReadWrite})$ from servers, where $g : \{0,1\}^B \to \{0,1\}^B$, output $D[i]$ to $\mathcal{C}$ then update $D[i] := g(D[i])$ internally.

Figure 4: Functionality of ORAM in the client-server setting.

associated with the path $\mathsf{PRF}_k(i)$. It also fills dummy blocks to the remaining slots in $T$. $\mathcal{C}$ will abort if there is no empty slot when appending a block.

After the local initialization of $T$, $\mathcal{C}$ outsources all blocks in $T$ to servers through $\mathcal{F}_{\mathsf{DowPIR}}$, and publishes $k$.

**Public Access.** To read a logical address $i$, parties compute all bucket indices $r_0, \cdots, r_L$ as well as the additional bucket index $r_{L+1}$ corresponding to $\mathsf{PRF}_k(i)$. Then servers return all index and valid fields of blocks in these buckets via $\mathcal{F}_{\mathsf{DowPIR}}$ to $\mathcal{C}$. $\mathcal{C}$ identifies the block storing the element $D[i]$ from the invariant of $T$, and retrieves the data field of that block as $d$ from servers via $\mathcal{F}_{\mathsf{DowPIR}}$.

To update the logical address $i$ as $D[i] := g(D[i])$, our protocol utilizes the additional bucket $\tilde{\mathcal{B}}_{r_{L+1}}$ to eliminate the need for an eviction procedure, making the protocol significantly more lightweight. Specifically, $\mathcal{C}$ first updates the valid field of all blocks in $(\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L})$ via $\mathcal{F}_{\mathsf{DowPIR}}$, by setting the valid field of all blocks with $\mathsf{index} = i$ to 0 while leaving other blocks unchanged. Then $\mathcal{C}$ identifies the block with $\mathsf{index} = i$ in $\tilde{\mathcal{B}}_{r_{L+1}}$ and updates its data field to $g(d)$ via $\mathcal{F}_{\mathsf{DowPIR}}$. The invariant is maintained after public access, since the updated block in $\tilde{\mathcal{B}}_{r_{L+1}}$ becomes the only real block with $\mathsf{index} = i$ in buckets on the path $\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L}, \tilde{\mathcal{B}}_{r_{L+1}}$.

**Private Access.** To read a logical address $i$ privately, the only difference compared to public access is to let $\mathcal{C}$ to retrieve the index and valid fields of blocks in the buckets privately via $\mathcal{F}_{\mathsf{DowPIR}}$, and all other steps are the same as public access.

To update the logical address $i$ as $D[i] := g(D[i])$ privately, $\mathcal{C}$ retrieves all blocks in the root bucket $\mathcal{B}_1$ from servers, updates the valid field of blocks with $\mathsf{index} = i$ to 0 to maintain the invariant, then inserts a new real block $(1, i, \mathsf{PRF}_k(i), g(D[i]))$ to substitute a dummy block in the root bucket, and finally writes the updated root bucket $\mathcal{B}_1$ back to servers via $\mathcal{F}_{\mathsf{DowPIR}}$. $\mathcal{C}$ will abort if there is no dummy block in the root bucket.

To avoid $\mathcal{C}$ aborting, similar to other tree-based ORAMs, parties run an eviction algorithm $\mathcal{G}_{\mathsf{evict}}$ (discussed later) after each read-then-write access.

**Eviction.** We adapts Path ORAM's eviction algorithm [SvDS+13] in a deterministic schedule [RFK+15] as the eviction algorithm $\mathcal{G}_{\mathsf{evict}}$ in our protocol. Specifically, $\mathcal{G}_{\mathsf{evict}}$ first marks each real block in an ordinary bucket along the eviction path to a dummy block, if there exists another real block with the same index field in a bucket with a smaller level of the tree. Then, it executes the eviction algorithm presented in the Path ORAM on all ordinary buckets along the path.

Each time after a private read-then-write access, parties select a path of the tree in the reverse lexicographic order and send all ordinary buckets on this path to $\mathcal{C}$. Then, $\mathcal{C}$ executes $\mathcal{G}_{\mathsf{evict}}$ on all buckets of this

**Protocol $\Pi_{\mathsf{CS\text{-}ORAM}}$ *(Part I)***

**Parameters:** Let $N$ be the number of logical addresses, $L$ be the depth of the tree, $L^{\mathsf{aux}}$ be the level to place the additional buckets, $Z^{\mathsf{stash}}$ be the number of blocks in the root bucket $\mathcal{B}_1$, $Z$ be the number of blocks in non-root ordinary buckets $\mathcal{B}_2, \cdots, \mathcal{B}_{2^{L+1}-1}$, $Z^{\mathsf{aux}}$ be the number of blocks in additional buckets $\tilde{\mathcal{B}}_0, \cdots, \tilde{\mathcal{B}}_{2^{L^{\mathsf{aux}}}-1}$, $\mathsf{PRF}$ be a keyed pseudorandom function: $\{0,1\}^{\lambda} \times \{0,1\}^{\lceil \log N \rceil} \to \{0,1\}^{L}$.

**Initialize:** On $(\mathsf{init}, D)$ from $\mathcal{C}$ and $(\mathsf{init})$ from servers,

1. $\mathcal{C}$ samples $k \leftarrow \{0,1\}^{\kappa}$ and sends $k$ to all servers.
2. For each $i \in [N]$, $\mathcal{C}$ computes indices $r := \mathsf{PRF}_k(i)$ and $r^{\mathsf{aux}} := \lfloor r/2^{L-L^{\mathsf{aux}}} \rfloor$, and appends a block $(1, i, r, D[i])$ to $\tilde{\mathcal{B}}_{r^{\mathsf{aux}}}$. $\mathcal{C}$ aborts when there is no empty slot when appending a block.
3. $\mathcal{C}$ fills all buckets $(\mathcal{B}_1, \ldots, \mathcal{B}_{2^{L+1}-1}, \tilde{\mathcal{B}}_0, \ldots, \tilde{\mathcal{B}}_{2^{L^{\mathsf{aux}}}-1})$ with dummy blocks.
4. Parties send $(\mathsf{init})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to outsource $T$ to servers, and initialize $\mathsf{ctr} := 0$.

**Public read:** On $(\mathsf{PublicRead}, i)$ from parties,

1. Parties compute the path index $r := \mathsf{PRF}_k(i)$, the indices of buckets on path $r$ as $r_j := 2^j + \lfloor r/2^{L-j} \rfloor$ for $j \in [0, L]$, and the additional bucket index $r_{L+1} := \lfloor r/2^{L-L^{\mathsf{aux}}} \rfloor$.
2. Parties send $(\mathsf{PublicRead})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to let $\mathcal{C}$ obtain the $(\mathsf{valid}, \mathsf{index})$ tuple of each block in buckets $\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L}, \tilde{\mathcal{B}}_{r_{L+1}}$.
3. $\mathcal{C}$ identifies the block with $\mathsf{valid} = 1$ and $\mathsf{index} = i$ in $\mathcal{B}_{r_{L+1}}$ and denotes it as the $\ell^{\mathsf{aux}}$-th block.
4. $\mathcal{C}$ identifies the smallest $j \in [0, L]$ s.t. a block with $\mathsf{valid} = 1$ and $\mathsf{index} = i$ exists in $\mathcal{B}_{r_j}$, and denotes such block as the $\ell$-th block in $\mathcal{B}_{r_j}$.
5. Parties send $(\mathsf{PrivateRead})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to let $\mathcal{C}$ privately obtain $d$ as the $\mathsf{data}$ field of the $\ell$-th block in $\mathcal{B}_{r_j}$ if $j \le L$, or $\ell^{\mathsf{aux}}$-th block in $\tilde{\mathcal{B}}_{r_{L+1}}$ if $j = L+1$.
6. $\mathcal{C}$ outputs $d$.

**Public read then write:** On $(\mathsf{PublicReadWrite}, i, g(\cdot))$ from $\mathcal{C}$ and $(\mathsf{PublicReadWrite}, i)$ from servers,

1. Parties invoke the above public read procedure with input $i$, and keep all intermediate values.
2. $\mathcal{C}$ prepares a bit vector $\{0,1\}^{Z^{\mathsf{stash}}+Z \cdot L}$ representing the updated $\mathsf{valid}$ field for all blocks in $(\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L})$, setting the $\mathsf{valid}$ field of all blocks with $\mathsf{index} = i$ to 0 while leaving other blocks unchanged.
3. Parties send $(\mathsf{PublicWrite})$ to $\mathcal{F}_{\mathsf{DowPIR}}$, to update the $\mathsf{valid}$ field of blocks in $(\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L})$ according to this vector.
4. $\mathcal{C}$ sends $\ell^{\mathsf{aux}}$ to all servers. Parties send $(\mathsf{PublicWrite})$ to $\mathcal{F}_{\mathsf{DowPIR}}$, to update the $\mathsf{data}$ field of the $\ell^{\mathsf{aux}}$-th block in $\mathcal{B}_{r_{L+1}}$ to $g(d)$.

Figure 5: Client-server ORAM protocol in the $\mathcal{F}_{\mathsf{DowPIR}}$-hybrid model (Part I).

path and writes the updated buckets back to servers.

We present our protocol $\Pi_{\mathsf{CS\text{-}ORAM}}$ formally in Figure 5, and the security of our protocol in Theorem 1 as:

**Theorem 1.** *For any number of servers, protocol $\Pi_{\mathsf{CS\text{-}ORAM}}$ (Figure 5) securely realizes functionality $\mathcal{F}_{\mathsf{CS\text{-}ORAM}}$ (Figure 4) against a semi-honest adversary in the $\mathcal{F}_{\mathsf{DowPIR}}$-hybrid model. $\Pi_{\mathsf{CS\text{-}ORAM}}$ incurs a bandwidth blowup of $O(1)$ for public access and $O(\log N)$ for private access, provided there exists a protocol that securely realizes $\mathcal{F}_{\mathsf{DowPIR}}$ with a constant bandwidth blowup for a sufficiently large block size $B$.*

We provide a sketched intuition of the proof, by arguing the correctness, obliviousness, and bandwidth blowup of the protocol as follows.

**Protocol $\Pi_{\mathsf{CS\text{-}ORAM}}$** *(Part II)*

**Private read:** On $(\mathsf{PrivateRead}, i)$ from $\mathcal{C}$ and $(\mathsf{PrivateRead})$ from servers,

1. $\mathcal{C}$ computes the path index $r := \mathsf{PRF}_k(i)$, the indices of buckets on path $r$ as $r_j := 2^j + \lfloor r/2^{L-j} \rfloor$ for $j \in [0, L]$, and the additional bucket index $r_{L+1} := \lfloor r/2^{L-L^{\mathsf{aux}}} \rfloor$.
2. $\mathcal{C}$ sends $(\mathsf{PrivateRead})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to obtain the $(\mathsf{index}, \mathsf{valid})$ tuples of all blocks in $(\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L}, \tilde{\mathcal{B}}_{r_{L+1}})$.
3. $\mathcal{C}$ identifies the smallest $j \in [0, L]$ s.t. a block with $\mathsf{valid} = 1$ and $\mathsf{index} = i$ exists in $\mathcal{B}_{r_j}$, and denotes such block as the $\ell$-th block in $\mathcal{B}_{r_j}$. If it is not found, locate it as the $\ell$-th block in $\tilde{\mathcal{B}}_{r_{L+1}}$.
4. Parties send $(\mathsf{PrivateRead})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to let $\mathcal{C}$ privately obtain $d$ as the $\mathsf{data}$ field of the $\ell$-th block in $\mathcal{B}_{r_j}$ or $\tilde{\mathcal{B}}_{r_{L+1}}$.
5. $\mathcal{C}$ outputs $d$.

**Private read then write:** On $(\mathsf{PrivateReadWrite}, i, g(\cdot))$ from $\mathcal{C}$ and $(\mathsf{PrivateReadWrite})$ from servers,

1. Parties invoke the above private read procedure with $\mathcal{C}$'s input $i$, and keep all intermediate values.
2. Parties update $\mathsf{ctr} := (\mathsf{ctr} + 1) \bmod 2^L$, and identify buckets at the $\mathsf{ctr}$-th path in $T$ according to the reverse lexicographic order, as $\mathcal{B}_{\mathsf{ctr}_0}, \cdots, \mathcal{B}_{\mathsf{ctr}_L}$. Specifically, $\mathsf{ctr}_0 := 1$, and $\mathsf{ctr}_j := 2 \cdot \mathsf{ctr}_{j-1} + (\lfloor \mathsf{ctr}/2^{j-1} \rfloor \bmod 2)$ for $j \in [1, L]$.
3. Parties send $(\mathsf{PublicRead})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to output $\mathcal{B}_{\mathsf{ctr}_0}, \cdots, \mathcal{B}_{\mathsf{ctr}_L}$ to $\mathcal{C}$.
4. $\mathcal{C}$ updates the $\mathsf{valid}$ field of all blocks in $\mathcal{B}_{\mathsf{ctr}_0}$, setting it to 0 when $\mathsf{index} = i$ and leaving other blocks unchanged. Then $\mathcal{C}$ updates a dummy block in $\mathcal{B}_{\mathsf{ctr}_0}$ to $(1, i, r, g(d))$. If there is no dummy block in $\mathcal{B}_{\mathsf{ctr}_0}$, $\mathcal{C}$ aborts.
5. $\mathcal{C}$ runs the eviction algorithm $\mathcal{G}_{\mathsf{evict}}$ on $(\mathcal{B}_{\mathsf{ctr}_0}, \cdots, \mathcal{B}_{\mathsf{ctr}_L})$ locally.
6. Parties send $(\mathsf{PublicWrite})$ to $\mathcal{F}_{\mathsf{DowPIR}}$ to write the updated $(\mathcal{B}_{\mathsf{ctr}_0}, \cdots, \mathcal{B}_{\mathsf{ctr}_L})$ back to servers.

Figure 5: Client-server ORAM protocol in the $\mathcal{F}_{\mathsf{DowPIR}}$-hybrid model (Part II).

**Correctness.** We begin by arguing that the invariant of the data structure $T$ is always preserved unless $\mathcal{C}$ aborts. After the update in a public read-then-write access, the only real block with the input index across all accessed buckets is the updated block in the additional bucket, maintaining the invariant. In a private read-then-write access, prior to invoking the eviction algorithm, the only real block with the input index in the root bucket is the newly inserted block, also ensuring the invariant is maintained. The eviction algorithm converts all "outdated" real blocks on the eviction path to dummy blocks then moves the remaining real blocks along the path. The invariant is preserved during this process.

Then, we analyze the probability of $\mathcal{C}$ aborting. The event of $\mathcal{C}$ aborting during initialization corresponds to the scenario where the maximum load exceeds $Z^{\mathsf{aux}}$ when $N$ balls are placed into $2^{L^{\mathsf{aux}}}$ bins at random. By applying the standard balls-and-bins analysis, the probability of this event is negligible when the parameters are set as $L^{\mathsf{aux}} = \lceil \log N - \log \log N \rceil$ and $Z^{\mathsf{aux}} = O(\log N)$. The scenario in which $\mathcal{C}$ aborts due to a full root bucket has been analyzed in the Path ORAM paper [SvDS+13]. Leveraging the same analysis and parameters $L = \lceil \log N \rceil$, $Z = O(1)$ and $Z^{\mathsf{stash}} = O(\log N)$, the aborting probability can be bounded as negligible.

We conclude the correctness of $\Pi_{\mathsf{CS\text{-}ORAM}}$, as the invariant of $T$ is maintained with high probability.

**Obliviousness.** The obliviousness of the protocol is straightforward in the $\mathcal{F}_{\mathsf{DowPIR}}$-hybrid model, since all access patterns to the outsourced data structure $T$ are either from the server's known inputs or from the PIR.

**Bandwidth Blowup.** We analyze the asymptotic bound of bandwidth blowup in the protocol. Suppose the underlying PIR protocol has a bandwidth blowup of $O(1)$ when the block size $B = \Omega(B_{\mathsf{PIR}})$, where $B_{\mathsf{PIR}} = \mathsf{poly}(\lambda, \log N)$. With the parameters above and this PIR protocol, the bandwidth cost of public read-then-write access is $O(\log^2 N + B)$ and that of a private read-then-write access is $O(\log^2 N + \log N \cdot B)$. That

**Functionality $\mathcal{F}_{\mathsf{MPCwPIR}}$**

This functionality interacts with $m$ parties.

**Initialize:** On $(\mathsf{init}, [\![D]\!])$, where $D$ is an array of $N$ elements and each $D[i] \in \{0,1\}^B$ for $i \in [N]$, store $D$ internally.

**Public read:** On $(\mathsf{PublicRead}, i)$, output $[\![D[i]]\!]$.

**Public write:** On $(\mathsf{PublicWrite}, i, [\![d]\!])$ where $d \in \{0,1\}^B$, update $D[i] := d$ internally.

**Private read:** On $(\mathsf{PrivateRead}, [\![i]\!], \mathcal{I})$, output $[\![D[i]]\!]$ if $i \in \mathcal{I}$; otherwise abort.

Figure 6: Functionality of PIR in the MPC setting.

**Functionality $\mathcal{F}_{\mathsf{RAM}}$**

This functionality interacts with $m$ parties.

**Initialize:** On $(\mathsf{init}, [\![D]\!])$, where $D$ is an array of $N$ elements and each $D[i] \in \{0,1\}^B$ for $i \in [N]$, store $D$ internally.

**Public read:** On $(\mathsf{PublicRead}, i)$, output $[\![D[i]]\!]$.

**Public read then write:** On $(\mathsf{PublicReadWrite}, i, [\![\mathsf{aux}]\!], g)$ where $\mathsf{aux} \in \{0,1\}^*$ and $g : \{0,1\}^* \times \{0,1\}^B \to \{0,1\}^B$, output $[\![D[i]]\!]$ then update $D[i] := g(\mathsf{aux}, D[i])$.

**Private read:** On $(\mathsf{PrivateRead}, [\![i]\!])$, output $[\![D[i]]\!]$.

**Private read then write:** On $(\mathsf{PrivateReadWrite}, [\![i]\!], [\![\mathsf{aux}]\!], g)$ where $\mathsf{aux} \in \{0,1\}^*$ and $g : \{0,1\}^* \times \{0,1\}^B \to \{0,1\}^B$, output $[\![D[i]]\!]$ then update $D[i] := g(\mathsf{aux}, D[i])$.

Figure 7: Functionality of RAM-based MPC.

is, our mixed-mode ORAM protocol incurs a bandwidth blowup of $O(1)$ for public access and $O(\log N)$ for private access when $B = \Omega(\max(B_{\mathsf{PIR}}, \log^2 N))$.

### 4.3 Extending to Secure Computation

We extend the client-server ORAM protocol from Section 4.2 to the secure computation setting, and build an MPC protocol in the RAM model. Under this model, we define the functionality supporting private and public accesses as $\mathcal{F}_{\mathsf{RAM}}$ in Figure 7.

The extended RAM-based MPC protocol on a high level is that all parties store the data structure $T$ through an MPC version of PIR functionality and jointly simulate the client $\mathcal{C}$'s algorithm (as circuits) through the black-box MPC functionality $\mathcal{F}_{\mathsf{ABB}}$. As in this setting, the circuit complexity is the primary metric for the efficiency of secure computation, we introduce some optimizations as follows and discuss the optimality of the protocol.

**MPC Version of PIR.** Similar to the PIR functionality in the client-server protocol, the RAM-based MPC protocol requires a PIR functionality in the MPC setting $\mathcal{F}_{\mathsf{MPCwPIR}}$ to build upon. It allows parties to securely fetch the corresponding data block using an index in a secret-shared manner. It also supports accessing data blocks using a public-known index. The functionality $\mathcal{F}_{\mathsf{MPCwPIR}}$ is described in Figure 6, and parties will store the data structure $T$ through it.

**Initialization.** After sampling a PRF key $k$ that is open to all parties, each party can initializes a secret sharing of $T$ using its own secret sharing of the data array $D$ locally, as $T$ is constructed deterministically

from the key $k$. Then parties store the secret-shared $T$ through the PIR functionality $\mathcal{F}_{\mathsf{MPCwPIR}}$ to enable future accesses.

**Access.** The RAM-based MPC protocol follows the same step as the client-server protocol for access in the secure computation setting as: parties interacts with the $\mathcal{F}_{\mathsf{MPCwPIR}}$ instead of $\mathcal{F}_{\mathsf{DowPIR}}$ in $\Pi_{\mathsf{CS\text{-}ORAM}}$ to access $T$, locally computes all steps with public-known inputs, and interacts with $\mathcal{F}_{\mathsf{ABB}}$ to simulate $\mathcal{C}$'s computation with private inputs.

**Eviction.** Regarding to the eviction algorithm $\mathcal{G}_{\mathsf{evict}}$, it follows the same steps as the client-server protocol, to mark all "outdated" blocks on the path as dummy blocks. Then, we substitute the eviction algorithm in Path ORAM to that in Circuit ORAM [WCS15], to achieve a better circuit complexity. We note that parameters in Circuit ORAM's eviction algorithm follow the same asymptotic bound as that in Path ORAM to achieve the same security level.

Putting all together, we summarize our result as Theorem 2.

**Theorem 2.** *For any number of parties, there exists a protocol that securely realizes functionality $\mathcal{F}_{\mathsf{RAM}}$ (Figure 4) against a semi-honest adversary in the $(\mathcal{F}_{\mathsf{MPCwPIR}}, \mathcal{F}_{\mathsf{ABB}})$-hybrid model. That protocol incurs a total circuit complexity of $O(B)$ for public access and $O(B \log N)$ for private access in $\mathcal{F}_{\mathsf{ABB}}$, provided there exists a protocol that securely realizes $\mathcal{F}_{\mathsf{MPCwPIR}}$ with a circuit complexity of $O(B)$ gates per access.*

**Analysis.** The correctness and obliviousness of the protocol are straightforward implied by the client-server results, and we only analyze the circuit complexity of the protocol in this part.

During a public read-then-write access, interaction with $\mathcal{F}_{\mathsf{MPCwPIR}}$ occurs a circuit complexity of $O(\log^2 N + B)$, and interaction with $\mathcal{F}_{\mathsf{ABB}}$ to simulate $\mathcal{C}$ occurs a circuit complexity of $O(\log^2 N)$, counting to a total circuit complexity of $O(\log^2 N + B)$.

The total circuit complexity of a private read-then-write access (except the eviction steps) is the same as that of a public read-then-write access. To evict a deterministic path, parties need to interact with $\mathcal{F}_{\mathsf{MPCwPIR}}$ to get all blocks on the path, leading to a circuit complexity of $O(\log^2 N + \log N \cdot B)$. Then, parties evaluate $\mathcal{G}_{\mathsf{evict}}$ in $\mathcal{F}_{\mathsf{ABB}}$. It takes $O(\log^3 N)$ gates to mark all outdated blocks to dummy blocks in a brute-force way, and the eviction algorithm in Circuit ORAM takes a circuit complexity of $O(\log N \cdot B)$ when $B = \Omega(\log^2 N)$.

Suppose the underlying protocol that realizes $\mathcal{F}_{\mathsf{MPCwPIR}}$ has a constant circuit complexity when the block size $B = \Omega(B_{\mathsf{PIR}})$, where $B_{\mathsf{PIR}} = \mathsf{poly}(\lambda, \log N)$. When taking $B = \Omega(\max(B_{\mathsf{PIR}}, \log^2 N))$, our RAM-based MPC protocol has a circuit complexity of $O(B)$ per public access and $O(\log N \cdot B)$ per private access.

# 5 Efficient Three-Party Mixed-Mode ORAM

In this section, we instantiate a concretely efficient mixed-mode RAM-based three-party computation (RAM-based 3PC) protocol in an honest-majority setting against a semi-honest adversary. We introduce optimizations to improve the concrete efficiency of the protocol, including an optimized DPF-based PIR in the 3PC setting for the tree structure and a new paralleled eviction algorithm.

## 5.1 ABB Functionality Instantiation

As discussed in Section 3.2, in the honest majority three-party setting, we can instantiate the $\mathcal{F}_{\mathsf{ABB}}$ functionality using a replicated secret sharing scheme [CDI05]. This scheme enables parties to perform a generic circuit evaluation using either a general $(3, 1)$-replicated MPC protocol [AFL+16], or a $(3, 1)$-garbled circuit protocol [MR18]. Based on the replicated secret-sharing scheme, we customize a variant of the $(3, 1)$-

replicated MPC protocol [AFL+16] for inner product computation. The parties locally XOR the intermediate evaluation results for each AND gate and defer the one-bit-per-gate communication cost to the end, reducing the overall communication cost to a single bit for evaluating the inner product of two bit vectors.

### 5.2 DPF-based PIR in 3PC Setting

We introduce a DPF-based PIR scheme in the 3PC setting to instantiate the $\mathcal{F}_{\mathsf{MPCwPIR}}$ functionality when $m = 3$. We recall the existing results [GI14, GKW18] to construct a two-server PIR from DPF, then propose our solution that leverages DPF to perform PIR on replicated shares, which eventually leads to our private path retrieval (PPR).

**DPF-based Two-server PIR.** Assume there are two servers $S_0$ and $S_1$ each has an identical copy of the array $D$ of size $N$ (in an encrypted form). When the client $\mathcal{C}$ wants to read the $i$-th element of $D$ without revealing $i$, it compute the DPF keys $(k_0, k_1) := \mathsf{DPF.Gen}(i)$ and sends $k_b$ to $S_b$ for each $b \in \{0, 1\}$. Then, each server $S_b$ computes $\boldsymbol{t}_b := \mathsf{DPF.EvalAll}(k_b)$ and $d_b := \oplus_{j\in[N]}(t_b^{(j)} \cdot D[j])$. We note that $d_0 \oplus d_1 = D[i]$ and the client can reconstruct $D[i]$ from outputs $d_0$ and $d_1$ from servers.

**DPF in Preprocessing Model.** We can extend the DPF-based PIR scheme to support a preprocessing phase such that only lightweight operations are required in the online phase. The client and servers can perform the key generation of DPF with a random index $r$ in the preprocessing phase and later correct the index to $i$ during the online phase. Specifically, during the preprocessing phase:

1. Client $\mathcal{C}$ samples a random $r \leftarrow [N]$, computes $(k_0, k_1) := \mathsf{DPF.Gen}(r)$, and sends $k_b$ to $S_b$ for $b \in \{0, 1\}$.
2. Server $S_b$ computes $\boldsymbol{v}_b := \mathsf{DPF.EvalAll}(k_b)$ for $b \in \{0, 1\}$.

During the online phase when $\mathcal{C}$ having $d := i - r$:

1. $\mathcal{C}$ sends the difference $d := (i - r) \bmod N$ to both servers.
2. Each server $S_b$ computes $\boldsymbol{t}_b := \mathsf{RightRotate}(\boldsymbol{v}_b, d)$ and $d_b := \oplus_{j\in[0,N)}(t_b^{(j)} \cdot D[j])$ for $b \in \{0, 1\}$.
3. Each $S_b$ sends $d_b$ to $\mathcal{C}$ for $b \in \{0, 1\}$ and $\mathcal{C}$ reconstructs $D[i] := d_0 \oplus d_1$.

**Moving it to 3PC Setting.** We can move the above two-server PIR scheme to the three-party setting, and leverage the preprocessing-online feature to avoid evaluating $\mathsf{DPF.Gen}$ using generic MPC. Consider the case that array $D$ of size $N$ is shared among three parties $P_0$, $P_1$, and $P_2$ in a replicated secret sharing scheme. That is, each Party $P_b$ holds two arrays $D_{b-1}, D_{b+1}$ such that $D[i] = D_0[i] \oplus D_1[i] \oplus D_2[i]$ for each $i \in [N], b \in \mathbb{Z}_3$. It is the case that both $P_{b-1}$ and $P_{b+1}$ hold an identical copy of the array $D_b$, and no encryption is required as there is no information leaked from the replicated shares. Therefore, $P_b$ can act as a client to privately obtain $D_b[i]$ by invoking the DPF-based PIR scheme without revealing $i$.

To do so, $P_b$ firstly computes the DPF key on a random position $r$ and sends the key to the other two parties in a preprocessing phase. All parties jointly evaluate the difference $(i - r) \bmod N$ through a generic MPC $\mathcal{F}_{\mathsf{ABB}}$ and reveal this difference to $P_{b-1}$ and $P_{b+1}$. In this way, the protocol lets $P_b$ obtain $D_b[i]$ without revealing $i$ to any party.

We note that $(D_0[i], D_1[i], D_2[i])$ is a one-out-of-three XOR secret sharing of $D[i]$, and parties can reshare $D[i]$ to a replicated secret sharing with one round of communication.

**Instantiation of $\mathcal{F}_{\mathsf{MPCwPIR}}$.** With the above technique, to instantiate the $\mathcal{F}_{\mathsf{MPCwPIR}}$ functionality in the 3PC setting, all three parties hold replicated secret shares of the array $D$. It is straightforward to perform public accesses on the shared array. Parties can perform private accesses using the above DPF-based PIR scheme, with $O(\log N)$ circuit complexity to compute the difference and $O(B + \log N \cdot \lambda)$ communication overhead using existing efficient DPF schemes.

**Private Path Retrieval.** We note that the private path retrieval technique proposed by Gordon et al. [GKW18] can be applied to this PIR protocol to reduce the number of DPFs required.

Let $L$ be the depth of the tree and $r$ be the path index. With the above PIR protocol, each party $P_b$ essentially needs to locally compute DPF keys and jointly evaluate the difference $L+1$ times to let parties $P_{b-1}$ and $P_{b+1}$ obtain an XOR secret sharing of unit vectors $\mathbf{unit}(2^L, r), \mathbf{unit}(2^{L-1}, \lfloor r/2 \rfloor), \cdots, \mathbf{unit}(2^0, \lfloor r/2^L \rfloor)$. Then each unit vector is used to retrieve the corresponding bucket along the path $r$.

However, we can apply the private path retrieval technique to reduce the number of DPFs required to only one. With an XOR secret sharing $\boldsymbol{t}_{L,b}$ of a unit vector $\mathbf{unit}(2^L, r)$ for $b \in \{0, 1\}$, such that $\boldsymbol{t}_{L,0} \oplus \boldsymbol{t}_{L,1} = \mathbf{unit}(2^L, r)$, parties can locally derive $\boldsymbol{t}_{L-1,b}$ from $\boldsymbol{t}_{L,b}$ by XORing the nearby elements, as $t^{(j)}_{L-1,b} := t^{(2j)}_{L,b} \oplus t^{(2j+1)}_{L,b}$ for $j \in [2^{L-1}]$. The property that $\boldsymbol{t}_{L-1,0} \oplus \boldsymbol{t}_{L-1,1} = \mathbf{unit}(2^{L-1}, \lfloor r/2 \rfloor)$ follows. We can repeat this process to derive all $\boldsymbol{t}_{j,b}$ for $j \in [0, L]$ locally.

This optimization reduces the amortized bandwidth cost of PIR from $O(\kappa L + B)$ to $O(\kappa + B)$ per element when retrieving all elements along a path on a tree with depth $L$.

### 5.3 Paralleled Eviction Algorithm

The eviction algorithm of Circuit ORAM provides an (almost) optimal circuit complexity asymptotically. However, there is a large concrete overhead when running it in the MPC setting. Inspired by the idea of Onion ORAM [DvDF+16, CCR19], we adjust the eviction frequency $A = O(\rho)$, such that the new paralleled eviction algorithm $\mathcal{G}_{\mathsf{evict}}$ will be invoked after $A$ times of private read-then-write operations. It significantly reduces the number of eviction operations, and correspondingly, we need to increase the bucket size $Z = Z^{\mathsf{stash}} = O(\rho)$ to bound the failure probability.

Let $\mathcal{B}_{r_0}, \cdots, \mathcal{B}_{r_L}$ denote the buckets along the eviction path from the root to the leaf, the eviction algorithm is parallelized by, for each $j \in [0, L)$, evicting all real blocks in $\mathcal{B}_{r_j}$ to one of its children buckets ($\mathcal{B}_{r_{j+1}}$ or its sibling $\mathcal{B}_{r_{j+1} \oplus 1}$) based on the $\mathsf{path}$ field of each block.

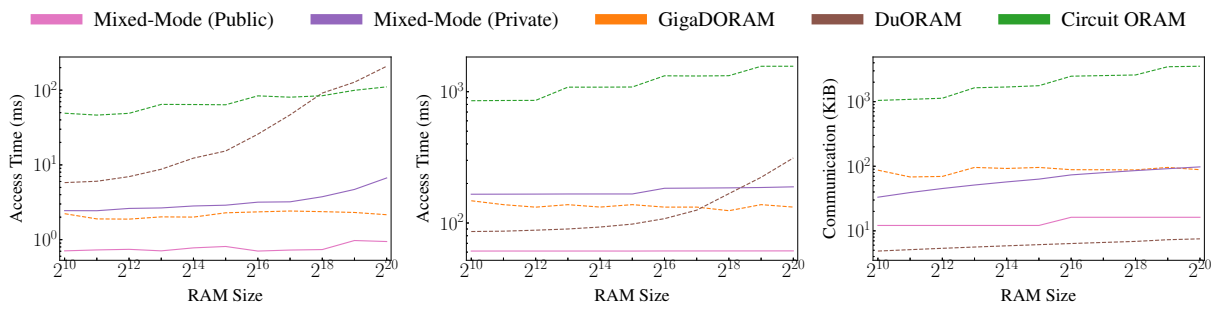
Specifically, for a real block $(1, i, r, d)$ in a non-leaf bucket $\mathcal{B}$, if the left child bucket of $\mathcal{B}$ resides on the $r$-th path, it will be moved into the left child bucket of $\mathcal{B}$; otherwise, it will be moved into the right child bucket. Any existing real blocks in the target child bucket with the same index will be updated to dummy blocks to maintain the invariant of data structure $T$.

To run the eviction algorithm in the MPC setting in a parallelized manner, for each $j \in [0, L)$, parties split the real blocks in $\mathcal{B}_{r_j}$ into two buckets based on each block's $\mathsf{path}$ field, and merge the two buckets with the corresponding child bucket. To merge two buckets, parties first get a $2Z$-sized list of blocks by concatenating the split parent bucket and the target child bucket, and have a stable sorting to all blocks in the list using $\mathcal{F}_{\mathsf{ABB}}$ based on the $(\mathsf{valid}, \mathsf{index})$ tuple in descending order. Then, they can update "outdated" blocks in the target child bucket to dummy blocks by updating the latter to dummy when there are two adjacent blocks with the same index using $\mathcal{F}_{\mathsf{ABB}}$. Parties sort the blocks again based on the $\mathsf{valid}$ field in descending order and retain the first $Z$ blocks as the merged bucket. For each $j \in [0, L-1)$, the child bucket of $\mathcal{B}_{r_j}$ along the eviction path will be empty, thus only one merge operation is required for each level, and parties can run the eviction algorithm with $L+1$ merge operations in parallel. The correctness of the eviction algorithm is guaranteed when there is no overflow after merge operations. We can bound this probability to be negligible when $Z = Z^{\mathsf{stash}} = O(\rho)$.

### 5.4 Putting All Together

We put all the components together to provide an overview of our RAM-based 3PC protocol here.

**Public access.** To read an element with index $i$ publicly, all parties can compute the path locally and access the corresponding buckets. Then, instead of getting the position of the target block, parties rather get a unit vector indicating whether it is the target block in a replicated-sharing form through $\mathcal{F}_{\mathsf{ABB}}$. Specifically, this step can be done by first computing a bit vector indicating whether each block has $\mathsf{valid} = 1$ and




(a) Access time in LAN (b) Access time in WAN (c) Communication load

Figure 8: **Wall-clock access time and communication load per read-then-write access when RAM size ranging from $2^{10}$ to $2^{20}$ entries of $8$ bytes.** Time unit is milliseconds (ms) and communication unit is kibibytes (KiB). Solid lines are the results of our protocol in different modes; dashed lines are the results of prior works.

$\mathsf{index} = i$, through a replicated MPC protocol, then only keeping the first non-zero bit through a garbled circuit protocol to minimize the concrete overhead. With this unit vector, parties can obtain the $\mathsf{data}$ field from an inner product computation. To update an element, parties can utilize the bit vector computed to mark all blocks with the same index as dummy blocks. Then they write an updated block to the corresponding position in the additional bucket as $i$ is known to all parties.

**Private access.** To privately read an element with index $i$, parties firstly compute the corresponding path through $\mathcal{F}_{\mathsf{ABB}}$, and applying the DPF-based PIR (PPR) mentioned above to obtain all buckets on the path. Then parties can use the same way as public access to get the $\mathsf{data}$ field of the target block from the secret-shared buckets on the path. To update an element, parties can use the same way to mark all blocks in the root bucket with the same index as dummy blocks (as the root bucket is on every path), and then write the updated block to the root bucket. Parties run the paralleled eviction algorithm $\mathcal{G}_{\mathsf{evict}}$ after privately updating $A$ elements to have a bounded failure probability.

# 6 Evaluation

In this section, we study the concrete performance of our mixed-mode RAM-based 3PC protocol with a focus of the following questions.

**Q1** How much improvement does our mixed-mode protocol make in efficiency when compared to prior works?

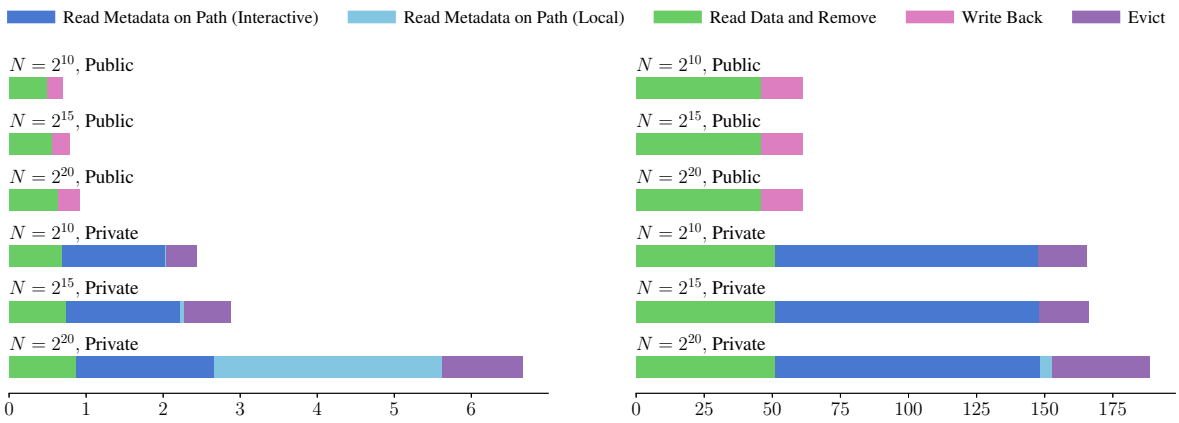
**Q2** What is the bottleneck of our protocol in different modes and network conditions of a moderate-size RAM?

**Q3** How much do the oblivious algorithm evaluations speed up after introducing the mixed-mode feature into RAM-based MPC?

We first introduce the experimental setup, and then address the above questions separately in three subsections.

## 6.1 Experimental Setup

We implement our instantiated RAM-based 3PC protocol in C++ on top of the EMP toolkit [WMK16] and the open-source implementation of GigaDORAM [FOSZ23]. All code is compiled using `GNU GCC 11.4.0`, with `-O3` flag enabled. For instantiation, we set the bucket size $Z = 128$ and the eviction




(a) Running time in LAN (ms)

(b) Running time in WAN (ms)

Figure 9: **Running time breakdown of a read-then-write access in different scenarios.** $N$ denotes RAM size. Elements in the RAM are size of 8 bytes. The running time of each stage is averaged from a sufficiently large number of access , which is a multiple of the eviction frequency $A$.

frequency $A = 96$ to achieve a security level of $2^{-40}$ failure probability. We instantiate the PRF using LowMC for mapping each index to a path [ARS+15].

All of our experiments are performed on three identical `r8g.large` AWS EC2 instances in the same availability zone, each with 2 vCPUs and 16 GiB memory. We measure the network latency between these instances as $0.132 \pm 0.006$ ms.

To fairly compare our protocol with previous works, we re-benchmark prior results as well as our protocol under the same benchmark platform, with all possible compiler optimization options enabled, and allow all these protocols to utilize a single thread of the machine. If not specified otherwise, we only measure the cost of read-then-write access in $\mathcal{F}_{\mathsf{RAM}}$ each protocol realizes, as all kinds of RAM access can be considered as read-then-write accesses. We range the RAM size from $2^{10}$ to $2^{20}$ elements, each element size of 8 bytes, corresponding to up to 8 MiB data in total.

We evaluate all these protocols in both a local area network (LAN) setting and a wide area network (WAN) setting. We execute them between three AWS instances without additional traffic control for the LAN setting. To simulate a WAN setting, we limit the bandwidth to 500Mbps and the round-trip time between any pair of instances to 10ms using the `tc` command.

## 6.2 Cost Comparison

To understand the overall performance of the mixed-mode protocol compared with prior state-of-the-art semi-honest RAM-based 3PC protocols, we compare the cost of our protocol in both public and private modes with the cost of GigaDORAM, DuORAM, and 3P-Circuit ORAM. GigaDORAM [FOSZ23] is a hierarchical-based RAM-based 3PC protocol with a fascinating performance when three parties are under a cluster placement. DuORAM [VHG23] that realizes the same functionality shows great performance in restricted networks. 3P-Circuit ORAM [JW18] is an optimized RAM-based 3PC protocol based on Circuit ORAM [WCS15]. As some protocols need cost amortization, we collect the evaluation results after running a sufficiently large number of accesses.

**Execution Time.** We evaluate the wall-clock execution time per access for each protocol when performing a sufficiently large number of read-then-write accesses in both the LAN and the WAN settings. We take the maximum time needed for all three machines as the results and show the results in Figure 8a and Figure 8b

respectively.

In the LAN setting, our protocol in public mode is consistently better than all the other protocols by at least $3\times$. Meanwhile, our protocol in the private mode introduces a roughly $50\%$ cost overhead over GigaDORAM on average, and is much more efficient than DuORAM and 3P-Circuit ORAM. We note that the gap between our protocol in the private mode and GigaDORAM broaden when the RAM size is greater than $2^{18}$ elements, as our protocol requires a linear complexity (local computation for private information retrieval) but GigaDORAM is sub-linear. However, for commonly used moderate database sizes, our protocol achieves a competitive performance and enables the feature of mixed-mode access.

In the WAN setting, our protocol in public mode outperforms GigaDORAM and DuORAM by roughly $2\times$, and is better than 3P-Circuit ORAM by orders of magnitude. Our protocol in private mode introduces an average $30\%$ overhead to GigaDORAM; however, this gap doesn't show a trend of broadening with the increase of the RAM size, as round complexity instead of local computation complexity overwhelms the overall performance in a restricted network setting. As DuORAM has a round complexity of a constant by batching operations during preprocessing, our protocol in private mode performs worse than DuORAM when the RAM size is less than $2^{18}$. Nevertheless, with a huge computation overhead of DuORAM, our protocol outperforms it when the RAM size is greater than $2^{18}$ since round complexity is not the bottleneck.

**Communication Overhead.** We evaluate the total communication sent by all parties per access through the network, for each protocol when performing a sufficiently large number of read-then-write accesses as figure 8c. It depicts the communication load of our protocol in both public and private modes is better than all the other protocols, with the only exception of DuORAM. As DuORAM utilizes the preprocessing phase with a relatively high computation overhead, our protocol provides a much better trade-off to improve practicality.

### 6.3 Microbenchmarks

To study the bottleneck of our mixed-mode protocol, we delve into the concrete cost of each stage under different network settings. Based on our protocol design and implementation, we divide the read-then-write access into $4$ sequential stages:

1. **Read Metadata on Path**: read the valid and index fields of blocks along the path, either privately or publicly depending on the access mode; this stage is further divided into an interactive part (mainly for communication) and a local part (mainly for computation);
2. **Read Data and Remove**: privately read the data field of the target block, and update all blocks with the same index field to dummy blocks along the path in public mode, or in the root bucket in private mode;
3. **Write Back**: write the updated block to the corresponding additional bucket in public mode, or the root bucket in private mode;
4. **Evict**: execute the eviction algorithm, applicable in private mode. The eviction cost is amortized over multiple accesses.

Figure 9 illustrates the running time for each stage of a read-then-write access in both LAN and WAN settings, as well as in both private and public modes. We consider RAM sizes of $2^{10}$, $2^{15}$, and $2^{20}$ elements, each of $8$ bytes, representing moderate-small, moderate, and moderate-large RAM sizes, respectively.

Figure 9a shows the time breakdown in the LAN setting. In this setting, the communication and round complexity do not significantly affect the results. Access in public mode enjoys strong scalability due to its sublinear computational complexity and constant bandwidth blowup. However, we notice that the local computation part of reading metadata on the path, specifically private path retrieval using generated DPF vectors, leads to a dramatic increase due to its linear computational complexity. It becomes the overall bottleneck when the RAM size is large.

Figure 9b depicts the breakdown in the WAN setting. In contrast to the LAN setting, communication and round complexity play a key role in the overall performance. In public mode, our protocol demonstrates

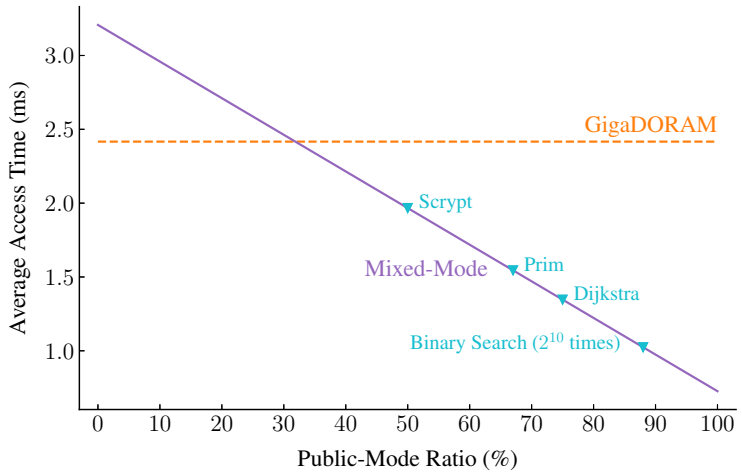



Figure 10: **Average wall-clock access time per RAM access under different public-mode ratios in the LAN setting.** The public-mode ratio indicates the percentage of RAM access in public mode over all RAM access in a RAM program. The size of each RAM element is 8 bytes and the RAM size is $2^{17}$ elements, corresponding to 1 MiB RAM in total.

strong scalability as the access time remains almost constant when the RAM size increases. This feature benefits from the constant bandwidth blowup of the protocol. In private mode, the bottleneck shifts to the heavy communication and round costs associated with evaluating circuits. Even when the RAM size is large, the local computation takes a smaller proportion of the overall cost in private mode.

### 6.4 Real-world Applications

To demonstrate the practicality of the mixed-mode feature and the benefit of lightweight public accesses, we prototype our protocol with several algorithms that are suitable or commonly used in RAM-based programs. We quantify the performance gain with respect to the public-mode ratio of each algorithm, which indicates the proportion of public-mode RAM access over the total number of RAM access. Firstly, we briefly introduce these algorithms in the following.

**Binary Search.** Our protocol allows an oblivious binary search on a secret-shared array. It requires $N$ public access to initialize the sorted array and $\log N$ private access for each binary search. For a moderate-size array of $N = 2^{17}$ elements with $k = 2^{10}$ queries of binary search, the public-mode ratio is $0.88$.

**Dijkstra.** The Dijkstra algorithm [Dij59] is a well-known single source shortest path algorithm. Given the secret-shared inputs graph $G = (V, E)$ and a source node $S$, our protocol allows private evaluation of the shortest path from $S$ to all other nodes. Specifically, the algorithm requires that the RAM stores the distance array, visitation status array, and adjacency matrix. Then, it accesses RAM in public mode for $3 \cdot (|V|^2 - |V|)$ times and in private mode for $|V|^2 + 1$ times, yielding a public-mode ratio of $0.75$ when $|V|$ is sufficiently large.

**Prim.** The Prim algorithm [Pri57] is an algorithm to find the minimum spanning tree of a graph. Assume that a private graph $G = (V, E)$ is represented as a secret-shared adjacency matrix stored in our RAM. To privately evaluate the minimum spanning tree of $G$, the Prim algorithm needs to access the RAM in public mode for $2 \cdot (|V|^2 - |V|)$ and in private mode for $(|V|^2 + |V|)$ times, leading to a public-mode ratio of $0.67$ when $|V|$ is sufficiently large.

**Scrypt.** Scrypt [PJ16] is a key derivation function with heavy memory usage to prevent brute-force attacks. Our protocol provides an efficient way to evaluate Scrypt in an MPC setting. The private evaluation of its

function ROMix requires $N$ public and $N$ private accesses where $N$ is the parameter of the cost factor, leading to a public-mode ratio of $0.5$.

Figure 10 shows the average RAM access time on each algorithm using our mixed-mode protocol as well as GigaDORAM in the LAN setting, based on the public-mode ratio of each application. Due to our lightweight public access protocol, the average access time is reduced with the increase of the public-mode ratio. We observe that our protocol outperforms GigaDORAM in all of these applications.

## References


[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

[AFN+17] Ittai Abraham, Christopher W. Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 91–120. Springer, Berlin, Heidelberg, March 2017.

[AHR24] Ananya Appan, David Heath, and Ling Ren. Oblivious single access machines - A new model for oblivious computation. In *ACM CCS 2024*, pages 3080–3094. ACM Press, November 2024.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Cham, May 2020.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Berlin, Heidelberg, April 2015.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Berlin, Heidelberg, April 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[BN16] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In Madhu Sudan, editor, *ITCS 2016*, pages 357–368. ACM, January 2016.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[CCMS19] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In Timothy M. Chan, editor, *30th SODA*, pages 2448–2467. ACM-SIAM, January 2019.

[CCR19] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 345–360. ACM Press, November 2019.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Berlin, Heidelberg, February 2005.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, Berlin, Heidelberg, August 2003.

[DvDF+16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Berlin, Heidelberg, January 2016.

[FOSZ23] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. GigaDORAM: Breaking the billion address barrier. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3871–3888. USENIX Association, August 2023.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Berlin, Heidelberg, May 2014.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GKW18] S. Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server ORAM. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 141–157. Springer, Cham, December 2018.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[GYW+23] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in COT and DPF. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 330–362. Springer, Cham, April 2023.

[JW18] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18International Conference on Applied Cryptography and Network Security*, volume 10892 of *LNCS*, pages 360–378. Springer, Cham, July 2018.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Berlin, Heidelberg, December 2014.

[LLG24] Xiang Li, Yunqian Luo, and Mingyu Gao. Bulkor: Enabling bulk loading for path ORAM. In *2024 IEEE Symposium on Security and Privacy*, pages 4258–4276. IEEE Computer Society Press, May 2024.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Cham, August 2018.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2292–2306. ACM Press, November 2021.

[MR18] Payman Mohassel and Peter Rindal. ABY$^3$: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*, pages 930–947. IEEE Computer Society Press, May 2022.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd ACM STOC*, pages 514–523. ACM Press, May 1990.

[PJ16] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. Technical report, 2016.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[RFK+15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 415–430. USENIX Association, August 2015.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Berlin, Heidelberg, December 2011.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[VHG23] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3907–3924. USENIX Association, August 2023.

[WCM18] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious RAM. *PoPETs*, 2018(4):64–84, October 2018.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 215–226. ACM Press, November 2014.

[ZE13] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507. IEEE Computer Society Press, May 2013.