DuetORAM: Two-Server Distributed ORAM with Constant Rounds and $O(\log N)$ Communication

Feng Li[1], Xiangfu Song[1,✉], Yingying Li[2], Lisha Yao[3], Guomin Yang[3],
Tianwei Zhang[1], Robert H. Deng[3]
[1]*Nanyang Technological University* [2]*Hangzhou Normal University*
[3]*Singapore Management University*
Emails: {li.feng,xiangfu.song,tianwei.zhang}@ntu.edu.sg, lyylyingying@163.com,
yaolishaqh@gmail.com, {gmyang,robertdeng}@smu.edu.sg, ✉ corresponding author

Abstract

Distributed Oblivious RAM (DORAM) is a promising building block for privacy-preserving cloud databases and outsourced storage systems. However, existing two-server designs often rely on slow linear scans or heavy cryptographic primitives, making them struggle to balance efficiency and bandwidth, and thus hindering their practical deployment.

We present DuetORAM, a two-server DORAM that achieves constant-round access with $O(\log N)$ communication while avoiding these computational bottlenecks. Our key idea is a replicated-to-shared block encoding that allows servers to keep identical ciphertexts for efficient PIR-based retrieval, while locally interpreting them as secret shares to enable oblivious eviction via a lightweight shuffle. We further design a secret-shared shuffle with an offline-online decomposition that shifts most bandwidth-intensive work to a preprocessing phase, significantly reducing online communication. We implement a prototype of DuetORAM and evaluate it under diverse network conditions. Our results show that DuetORAM outperforms both the state-of-the-art two-server scheme DUORAM (reducing retrieval latency by up to 170× in LAN settings), and three-server design S^{3}ORAM (reducing retrieval latency by 1.7× in LAN and accelerating eviction by 7× in LAN and 5× in WAN, respectively).

1 Introduction

Oblivious RAM (ORAM) [20] is a cryptographic primitive, which hides not only data contents but also access types and addresses. Such protection is crucial for metadata-hiding applications, where access traces alone may leak sensitive information even if the accessed data is encrypted. ORAM has been widely adopted in many scenarios. For instance, it has been used in encrypted search systems [5, 13, 17, 25, 27, 29, 31, 40] to defeat leakage-abuse attacks [6, 26, 41] over searchable encryption [12]. ORAM is also used in metadata-hiding file sharing [9, 10] and oblivious analytic systems [7, 28]. In practice, Signal [1] deploys an ORAM-based solution to protect users' phone numbers, and Cloudflare proposes an ORAM-based encrypted search system, UtahFS [2].

Despite its promising applications, ORAM is still relatively expensive for practical use. Classical client-server ORAM constructions incur polylogarithmic communication and computation overhead [3, 20, 37] per access. For instance, the conceptually simple *tree-based* ORAMs [32, 34, 37] organize blocks within buckets along a binary tree structure and support oblivious access by reading and rewriting a root-to-leaf path. As a result, each access incurs logarithmic client-server communication. Besides, ORAM requires *oblivious eviction* to maintain correctness and security. In the single-server setting, eviction typically requires the client to download the relevant blocks from the server, locally decrypt and reorganize them, re-encrypt the updated blocks, and upload them back. Eviction, either performed during each access [30] or amortized over multiple accesses [32], incurs significant bandwidth and latency overhead. This is particularly costly in the wide-area network setting where interaction and communication dominate performance. Though Onion-ORAM [14] demonstrates that it is theoretically possible to avoid logarithmic communication blowup via server-side computation, it requires expensive homomorphic encryption that incurs significant latency.

Distributed ORAM (DORAM) [15, 23, 24, 35] aims to reduce the overhead in client-server ORAM. DORAM shifts the client-server overhead to server-side secure computation between non-colluding servers. This trade-off is well motivated in practice: inter-server communication is often faster and more stable than client-server networks, and servers possess powerful computing resources. This *multi-server* model is particularly attractive as a deployment point: (1) it requires only a minimal non-collusion assumption, which has been accepted by other privacy-preserving applications and under standardization [19]; (2) it avoids the operational complexity of larger committees.

Several DORAM constructions [24] use three (or more) servers and share the ORAM storage among them. The extra servers enable redundancy from Shamir or replicated secret

sharing, which makes oblivious access and eviction efficient. In this case, oblivious access and eviction steps can be expressed using only lightweight operations without invoking expensive computation or scanning over the entire database. The price is a stronger deployment assumption that requires more non-colluding servers and, for several tree-based multi-server constructions [24], an additional communication structure during eviction (*e.g.*, layer-by-layer procedures with multiple rounds) to achieve $O(\log N)$ blocks of communication in $O(\log N)$ rounds, as detailed later. Table 2 in Appendix §A summarizes the theoretical costs of prior work.

Designing 2-server DORAMs with comparable efficiency is challenging, as 2-Party Computation (2PC) is generally more expensive than 3PC with an honest majority. To realize server-side oblivious eviction, existing works [15, 39] entail a linear-scan design. The construction Floram [15] achieves $O(\log N)$ communication in constant rounds but requires linear work to compute over the whole database per access. In addition, Floram periodically refreshes ORAM storage entirely after $O(\sqrt{N})$ accesses, yielding amortized $O(\sqrt{N})$ inter-server communication. Follow-up works [38, 39] follow a similar approach. Hamlin and Varia [22] demonstrate that 2-server DORAM can achieve constant rounds and sublinear server work. However, they follow the square-root ORAM [20] with $\tilde{O}(\sqrt{N})$ communication complexity, and adopt security relaxations that allow distinguishing between read and write operations. In addition, their reshuffling components rely on relatively heavy mechanisms without implementation, for which the concrete efficiency remains unclear.

To date, the following question remains open:

> *Can we design a concretely efficient 2-server DORAM with $O(\log N)$ blocks of inter-server communication in constant rounds and low client-server communication only using cheap symmetric primitives, and without linear server-side work?*

We propose DuetORAM, a 2-server DORAM scheme that enjoys $O(\log N)$ blocks of inter-server communication, while enabling efficient oblivious eviction using only lightweight symmetric primitives[1]. A key feature of our DuetORAM is the *co-design* of the ORAM block encryption and the primitives for oblivious eviction: the servers maintain replicated ciphertexts that are ideal for efficient 2-server Private Information Retrieval (PIR), yet the encrypted blocks can be locally converted as secret shares of the underlying plaintext blocks. This non-interactive *ciphertext-to-share* bridge enables oblivious eviction as an oblivious permutation over shared plaintext blocks, without first running an expensive interactive secret-shared decryption. To further reduce online bandwidth, we propose an efficient oblivious permutation protocol from oblivious secret-shared shuffle via a new offline-online decomposition, which is also of independent interest.

We implement DuetORAM and evaluate its efficiency under diverse network conditions. Our results show that, compared to the prior 2-server DORAM scheme DUORAM, DuetORAM reduces retrieval latency by up to 170× in LAN settings; compared to the 3-server design S^{3}ORAM, DuetORAM achieves 1.7× reduction in retrieval latency, while also accelerating eviction by 7× in LAN and 5× in WAN settings.

1.1 Technical Overview

We target a 2-server DORAM with low round complexity, low communication complexity, and concretely practical efficiency, without using heavy cryptographic mechanisms.

System model and high-level procedures. DuetORAM involves a lightweight client and two non-colluding servers S_0, S_1, adapting tree-based ORAM to the 2-server setting. Concretely, we follow the eviction strategy and parameters from [24], originally derived from Onion ORAM [14]. Servers store an identical copy of $O(N)$ ORAM (encrypted) blocks organized in a binary-tree structure, where $N \leq A \cdot 2^{H-1}$ (A is the eviction frequency). The client keeps a position map locally, recording where a block resides in this tree. Each bucket holds up to Z blocks. Following the access strategy of tree-based ORAM, each logical (read/write) access touches one root-to-leaf path of height $H = \lceil \log N/A \rceil + 1$, which contains at most $Z \cdot (H+1)$ encrypted blocks. After accessing a block, the client remaps the re-encrypted (and possibly updated) block to a fresh random leaf and inserts it back into the root. To prevent block overflow in the root, the client periodically selects an eviction path for eviction.

Threat model. In our design, we consider a trusted client and two non-colluding[2], semi-honest servers, denoted as S_0 and S_1. Our threat model follows the standard framework used in distributed secure computation and oblivious storage. A Probabilistic Polynomial-Time (PPT) adversary $\mathcal{A}$ can statically corrupt at most one of the two servers S_t ($t \in \{0,1\}$). Under the semi-honest assumption, the corrupted server faithfully follows the protocol specification but attempts to infer sensitive information by inspecting its internal view. The view of a corrupted server S_t includes its private Pseudorandom Function (PRF) key, the local encrypted storage, and the transcripts of all incoming messages from the client or the peer server during protocol execution.

1.1.1 Replicated Encrypted Blocks

DuetORAM features a block encryption mechanism that simultaneously supports efficient 2-server oblivious retrieval and oblivious eviction.

[1] Looking ahead, our claim applies to a non-recursive setting where the client locally stores the position map, the same setting as [14, 23, 24]. The overhead incurred by applying the recursion technique [34] to the position map is discussed in Section 3.6.

[2] This assumption is standard in the two-server DORAM setting and is well-motivated in practice. For example, the two servers may be operated by independent service providers or deployed in different administrative or legal jurisdictions.

In the setup phase, the client samples two random PRF keys k_0, k_1 and encrypts each plaintext block b via a nonce-based double-masking encryption mechanism, where fresh nonces η_0, η_1 are randomly sampled. A ciphertext is computed as

$$(\eta_0, \eta_1, c \leftarrow b \oplus F(k_0, \eta_0) \oplus F(k_1, \eta_1)),$$

where F is a PRF and S_0 knows k_0 and S_1 knows k_1. Both servers store the encrypted block (η_0, η_1, c). Clearly, the encrypted block reveals nothing to each server, as each server only learns one PRF key.

A nice property following the nonce-based block encryption is that the servers can locally convert an encrypted block and share the encrypted plaintext block b:

$$S_0 : b_0 \leftarrow F(k_0, \eta_0), \qquad S_1 : b_1 \leftarrow c \oplus F(k_1, \eta_1),$$

so that $b_0 \oplus b_1 = b$, meaning the servers share the plaintext b for free. Looking ahead, this ciphertext-to-share bridge is the key observation for the subsequent oblivious eviction protocol: the servers can operate on secret shares of plaintext blocks without running an expensive decryption protocol within secure computation.

• ***Oblivious block retrieval without server-side interaction***. Oblivious access within a path is simple, as both servers store identical ciphertexts, and a simple 2-server PIR [11] suffices. The client performs a 2-server PIR query and receives the target ciphertext c, without revealing which block was selected. The client locally recovers b by simply computing

$$b = c \oplus F(k_0, \eta_0) \oplus F(k_1, \eta_1).$$

Note that 2-server PIR is highly simple and efficient: the retrieval is constant-round and its client-to-server communication is linear with the path length rather than the full database, and the servers do not interact with each other.

• ***Oblivious eviction from oblivious permutation***. Now we turn to oblivious eviction. At a high level, eviction in tree-based ORAMs mainly conducts the following: 1) collecting blocks from eviction buckets, and 2) reassigning these blocks to buckets while ensuring correctness.

With our ciphertext-to-share bridge, the servers locally convert encrypted blocks to secret-shared plaintext blocks non-interactively. Hence, oblivious eviction reduces to a 2-server *secret-shared permutation*: given secret shares of the blocks, the servers obtain secret shares of the permuted blocks for an eviction-dependent permutation, without learning the permutation itself or the plaintext blocks. While oblivious eviction via oblivious permutation is a natural idea, realizing it efficiently in the 2-party setting is challenging. Before presenting our solution, we first review possible approaches.

A permutation matrix is the most conceptually simple way to realize a permutation. Specifically, the client computes the eviction permutation (using the ORAM position map) and shares the corresponding permutation matrix between the servers. The servers then apply the permutation by evaluating a secret-shared matrix-vector multiplication over the blocks in the eviction region. This approach can be realized in $O(1)$ rounds via 2PC, but it incurs $O(Z^2 \log^2 N)$ blocks of communication. Z and N can be relatively large in practice; the bandwidth cost is high.

A standard optimization is to decompose the eviction permutation into a sequence of smaller permutations, as done by the triplet-eviction procedure from [14, 24]. In this case, each sub-eviction operates on a smaller permutation matrix, reducing the total communication to $O(Z^2 \log N)$ blocks. The downside is increased interaction: the sub-evictions must be executed sequentially from higher to lower layers, so lower layers wait for the completion of higher-layer eviction. As a result, the protocol requires $O(\log N)$ rounds to complete.

1.1.2 Oblivious Permutation

We propose an efficient 2-party secret-shared oblivious permutation with linear communication and constant rounds. Using our oblivious permutation protocol, we achieve 2-server oblivious eviction over the whole eviction path with optimal $O(Z \log N)$ blocks of communication in $O(1)$ rounds. We also prove that our whole-path eviction strategy is equivalent to the triplet-eviction strategy [14, 24] (c.f., Lemma 2, Appendix §D), which is of independent interest.

Our efficient 2-party secret-shared permutation protocol is based on pre-distributed *shuffle correlations*, inspired by the CGP shuffle [8]. In the offline phase, the client generates correlated randomness associated with a set of randomly punctured indexes and distributes the resulting correlations to the two servers. Later, in the online phase, the servers leverage these correlations to permute secret-shared blocks while keeping both the permutation and the plaintext blocks hidden.

The pre-distributed correlations encode only a set of *random* indexes, whereas eviction requires a meaningful, data-dependent eviction permutation. We exploit a nice property of these shuffle correlations that allows the client to provide a small amount of adjustment information, enabling the servers to *adapt* the pre-distributed correlations from the random indexes to the expected eviction permutation, without revealing any additional information.

We further introduce a *local-stretch* optimization for shuffle correlations that is particularly suited to the DORAM setting. Roughly, the client generates shuffle correlations whose element size is *independent* of the ORAM block size. Receiving the correlations, the servers can then locally *stretch* these compact correlations via a Pseudorandom Generator (PRG) to support oblivious permutation over full-size ORAM blocks. This property is desirable for DORAM: the client's offline communication for distributing correlations no longer depends on the ORAM block size, slashing client-server communication, especially for DuetORAM with large blocks.

From shared blocks to encrypted blocks. A remaining step

after the oblivious eviction is to maintain the replicated encrypted ORAM blocks for future oblivious access. This can be done efficiently following our block encryption mechanism. For each shared block, with S_0 holding b_0 and S_1 holding b_1, S_0 derives a random nonce η_0, computes $c_0 \leftarrow b_0 \oplus F(k_0, \eta_0)$, and sends (η_0, c_0) to S_1. At the same time, S_1 derives a random nonce η_1, computes $c_1 \leftarrow b_1 \oplus F(k_1, \eta_1)$, and sends (η_1, c_1) to S_0. Both servers compute $c \leftarrow c_0 \oplus c_1$ and then hold the identical encrypted block (η_0, η_1, c).

1.1.3 Putting All Together

We combine the above techniques to design a 2-server DORAM scheme. Each logical operation proceeds as follows.

- The client performs 2-server PIR over a root-to-leaf path to retrieve the desired blocks, locally decrypts/updates the block, remaps it to a fresh leaf, and writes back the re-encrypted block (with fresh nonces) at the root.
- Periodically, the two servers perform eviction *without* client-side heavy work: they locally convert replicated ciphertexts into shares of plaintext blocks, run a constant-round secret-shared permutation using pre-distributed shuffle correlations (with a small online adjustment from the client), and finally re-encode the permuted shares back to replicated ciphertexts.

Overall, DuetORAM achieves constant rounds for access without inter-server communication, and eviction with optimal $O(\log N)$ blocks of communication in constant rounds.

1.2 Contributions

We summarize the main contributions as follows:

- **New 2-server DORAM design**. We propose DuetORAM, a 2-server DORAM from tree-based ORAM. It only relies on simple symmetric primitives without requiring linear-scan paradigms or expensive operations. DuetORAM enjoys 1) constant-round oblivious access without inter-server communication, and 2) $O(\log N)$ blocks of communication in $O(1)$ rounds. To our best knowledge, no existing 2-server DORAM scheme achieves the above properties in a practical sense.
- **New tailored techniques and optimizations**. DuetORAM cannot achieve its goal without tailored techniques and optimizations that carefully exploit the structural and task-specific properties. In particular, we highlight the following key techniques that may be of independent of interest.
  - **Replicated encrypted ORAM storage**. The servers store the identical encrypted storage for a tree-based ORAM. This enables oblivious access from a highly efficient 2-server PIR over the queried path in constant rounds and without inter-server interaction.
  - **Novel block encryption mechanism**. We propose a novel block encryption mechanism that supports locally conversion from encryption blocks to secret shares of the underlying plaintext blocks, avoiding interactive decryption. This allows the servers to perform eviction directly over secret-shared plaintext blocks.
  - **New oblivious permutation with offline-online decomposition**. We propose a new 2-party oblivious permutation. Our new permutation protocol features an offline-online decomposition property that is well-suited for 2-server ORAM. Plugging in the protocol for oblivious eviction, DuetORAM achieves eviction with $O(\log N)$ blocks of communication in $O(1)$ rounds. Note that even the existing 3-server DORAM [24] requires $O(\log N)$ rounds for eviction.
- **Implementation and evaluation.** Apart from the promising asymptotic efficiency properties, DuetORAM is concretely efficient. We have implemented DuetORAM and open-sourced the code. Performance evaluation over various network settings shows that DuetORAM is even more efficient than the existing 3-server DORAM schemes, which require stronger assumptions.

1.3 Related Work

Single-Server ORAMs. Classical client-server ORAMs typically incur polylogarithmic communication and computational overhead [3, 20, 37] per access. The seminal Square-Root ORAM [20] requires $O(\sqrt{N})$ client-server communication per access, alongside a periodic $O(N \log N)$ communication for its oblivious sorting. Due to the heavy sorting bottleneck, subsequent Hierarchical ORAM organizes blocks into levels of geometrically increasing sizes to amortize the sorting overhead to $O(\log^3 N)$. Partition ORAM [36] avoids sorting over the entire dataset by splitting a large ORAM into multiple smaller, independent ORAM instances. Moving away from hierarchical shuffles, tree-based ORAMs [32, 34, 37] organize blocks within buckets along a binary tree structure and support oblivious access by reading and rewriting a root-to-leaf path, reducing client-server communication to logarithmic blocks. Onion ORAM [14] leverages server-side computation to circumvent the logarithmic communication blowup, yet it suffers from substantial computational latency due to the heavy homomorphic encryption involved.

Distributed ORAMs. DORAMs [15, 23, 24, 35] shift the client-server bottleneck to server-side secure computation between non-colluding servers. Existing architectures [15, 21, 38] are heavily polarized by their system models. In the 2-server setting, schemes are predominantly hampered by linear server-side work. Floram [15] leverages function secret sharing and 2-server PIR to achieve $O(\log N)$ online communication in constant rounds, but it demands a full linear scan per access and introduces an amortized $O(\sqrt{N})$ inter-

server communication blowup for storage refreshes. DUORAM [38] optimizes Floram by introducing a preprocessing phase to reduce the online client-server communication to a constant number of blocks per access, yet it still yields linear server scans and considerable latency. Another representative 2-server tree-based design, GKWORAM [21], extends the 2-server PIR scheme to minimize client-side storage, but it still requires the servers to perform a linear scan over the database while incurring logarithmic communication.

To bypass the linear server-work barrier, several works [16, 24] introduce an honest majority with three or more servers. GigaDORAM [16] optimizes hierarchical solutions under a 3-server setting to achieve polylogarithmic computation and communication. S^3ORAM [24] utilizes three (or more) servers based on Shamir secret sharing to provide highly efficient operational performance, but its sequential, layer-by-layer eviction strategy requires $O(\log N)$ blocks of communication across $O(\log N)$ rounds.

By contrast, DuetORAM achieves a concretely efficient 2-server DORAM with $O(\log N)$ blocks of inter-server communication in constant rounds and low client-server communication only using cheap symmetric primitives, and without linear server-side work.

2 Preliminary

Notation. $x \xleftarrow{\$} \{0,1\}^n$ denotes that x is a n-bit string chosen uniformly at random. $|x|$ denotes the size of the string x. We use bold lowercase letters to represent vectors. For a vector $\boldsymbol{a}$, its i-th element is denoted by $\boldsymbol{a}[i]$. We use $[\![x]\!]$ to denote an XOR secret sharing of $x \in \{0,1\}^n$. In the two-party case, $[\![x]\!]$ denotes the party P_t holds a share $[\![x]\!]_t$ such that $x = [\![x]\!]_0 \oplus [\![x]\!]_1$. Table 1 summarizes some of the notations used in this paper.

Table 1: Summary of Notation

Notation	Description
T, NT	ORAM tree and nonce tree
pm, md	Position map and bucket metadata of the ORAM tree
A	Eviction frequency
N, Z	Number of blocks and bucket size
B, b	Block size and block
$\mathsf{T}[i,j]$	Block at the j-th position in the i-th bucket
$\mathsf{pID}, \mathsf{pIdx}, \eta$	Path ID, path index and nonce
$\mathcal{P}(\mathsf{pID})$	Ordered indexes of all buckets on the path pID
$\mathcal{P}(\mathsf{pID}, h)$	Index of the bucket on the path pID at level h
n_r, n_e	Counter for access and eviction operations
$[n]$	All integers from 1 to n
$[\![\cdot]\!]$	Additive linear secret sharing

2.1 Permutation

A permutation is a bijective function $\pi : [n] \mapsto [n]$, and $\mathbf{S}_n$ is the symmetric group containing all $[n] \mapsto [n]$ permutations. When applying π over a vector $\boldsymbol{x}$ of n elements, we have

$$\boldsymbol{y} = \pi(\boldsymbol{x}) = (\boldsymbol{x}_{\pi(1)}, \cdots, \boldsymbol{x}_{\pi(n)}), \tag{1}$$

where $\boldsymbol{y}_i = \boldsymbol{x}_{\pi(i)}$. Namely, $\boldsymbol{x}_{\pi(i)}$ is moved to position i. We denote π^{-1} as the inverse of a permutation π. We use $\pi = \pi_0 \circ \pi_1$ to denote that π as the composition of two sub-permutations π_0 and π_1. For $i \in [n]$, it is calculated as

$$\pi_0 \circ \pi_1(i) = \pi_0(\pi_1(i)). \tag{2}$$

2.2 Two-server Private Information Retrieval

Private Information Retrieval (PIR) enables the retrieval of a data item from a (public) database DB while hiding which item is being accessed. A two-server PIR scheme $\Pi_{\mathrm{PIR}} = (\mathsf{Create}, \mathsf{Retrieve}, \mathsf{Reconstruct})$ is defined as follows:

- $\mathsf{Create}(i)$: On input an index i, output queries $(\boldsymbol{q}_0, \boldsymbol{q}_1)$.
- $\mathsf{Retrieve}(t, \boldsymbol{q}_t, \mathsf{DB})$: On input a server index t, a query $\boldsymbol{q}_t$, and the public database DB, output an answer a_t.
- $\mathsf{Reconstruct}(a_0, a_1)$: On input query result a_0 and a_1, reconstruct the result a.

Correctness of PIR requires that the responses a_0 and a_1 from PIR servers should reconstruct the desired item $\mathsf{DB}[i]$, while security requires that any PPT adversary who corrupts only one server learns no more information about the client's query. We provide the formal definition in the Appendix §B.

2.3 CGP Secret-Shared Shuffle

A secret-shared shuffle protocol allows parties to obliviously shuffle a dataset via a random secret permutation and obtain additive secret shares of the result, without either party learning the underlying permutation or the shared data. Chase et al. [8] proposed a secret-shared shuffle protocol in the two-party setting that avoids reliance on costly public-key cryptography or securely evaluating permutation networks. They introduced the Share Translation protocol, which leverages Oblivious Transfer (OT) [4] and puncturable PRFs to generate two sets of pseudorandom numbers (or shuffle correlations), effectively transforming the data permutation problem into a pseudorandom number permutation problem. In what follows, we provide a concise overview of the CGP secret-shared shuffle protocol.

Assume that party P_0 holds a sub-permutation π_0, and party P_1 possesses a secret share $[\![\boldsymbol{v}]\!]_0$. Party P_1 randomly generates a full matrix, while P_0 obtains a punctured matrix through the OT protocol, where the element at position $(i, \pi_0(i))$ is omitted. Using the punctured matrix, P_0 can XOR all the elements in row i and column $\pi_0(i)$ to compute $\boldsymbol{\Delta}[i]$, while P_1 can XOR the elements in column i to calculate $\boldsymbol{a}[i]$ and XOR the elements in row j to compute $\boldsymbol{b}[j]$. Figure 1 illustrates the Share Translation protocol in a two-party setting.

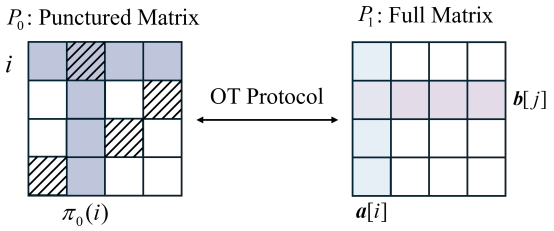


Figure 1: Share Translation Protocol

Based on the sets of pseudorandom numbers $(\boldsymbol{\Delta}, \boldsymbol{a}, \boldsymbol{b})$, P_1 sends $[\![\boldsymbol{v}]\!]_0 \oplus \boldsymbol{a}$ to P_0 and sets its share to $\boldsymbol{b}$, while P_0 computes its share as $\pi_0([\![\boldsymbol{v}]\!]_0 \oplus \boldsymbol{a}) \oplus \boldsymbol{\Delta}$. Next, with P_1 holding the sub-permutation π_1 and P_0 holding the share $[\![\boldsymbol{v}]\!]_1$, the operation is repeated. In the end, both parties obtain additive secret shares of $\pi_0 \circ \pi_1([\![\boldsymbol{v}]\!]_0 \oplus [\![\boldsymbol{v}]\!]_1)$.

2.4 Two-server Distributed Oblivious RAM

Definition 1 (Two-server DORAM)**.** *A two-server distributed ORAM scheme Π_{ORAM} consists of two protocols between a client and two non-colluding servers:*

- $\mathsf{Setup}(1^\lambda, \mathsf{DB})$*: On input a security parameter* λ *and a database* DB*, the client generates the initial encrypted storage structures* EDB_0 *and* EDB_1 *for the two servers and initializes its local state* σ*.*
- $\mathsf{Access}(\sigma, \mathsf{op}, \mathsf{id}, \mathsf{data})$*: This is an interactive protocol between the client and servers. The client inputs state* σ*,* $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$*, a block identifier* id*, and an optional data value* data*. If* $\mathsf{op} = \mathsf{read}$*, the protocol returns the block b identified by* id*. If* $\mathsf{op} = \mathsf{write}$*, the protocol returns the block b and updates the content of block* id *to* data*. Additionally, it updates the server-side structures to* EDB'_t *and client state to* σ'*.*

The formal definitions of correctness and security can be found in Appendix §C.

3 Construction of Our DuetORAM

We now present the concrete construction of DuetORAM. Figure 2 illustrates its overall workflow.

3.1 Roadmap & Challenges

Our goal is to efficiently *compile* a tree-based client-server ORAM to a 2-server DORAM. The core requirement is to reduce the client-side work and client-server communication and interaction, while keeping the server-side computation and inter-server communication as little as possible.

We start from a strawman construction to demonstrate why this is challenging. For any tree-based client-server ORAM, we can simply replicate its encrypted blocks between two servers. Oblivious access is straightforward then: the servers simply perform 2-server PIR over the encrypted blocks over the accessed path, which is highly efficient; it only requires non-interactive multiplication between a secret-shared one-hot vector and the encrypted blocks [11].

The challenge lies in the eviction process, which is the most expensive part and is performed periodically to maintain obliviousness and correctness. As motivated, we want the servers, rather than the client, to perform eviction via 2PC. Yet, eviction must be done securely without revealing the underlying plaintext blocks or eviction permutation. As the first step, the servers must somehow share the underlying plaintext blocks before conducting the secret-shared eviction procedure. If a normal encryption (*e.g.*, AES) is used, then sharing the plaintext would necessitate an interactive secret-shared decryption (*e.g.*, 2-party secure AES evaluation), which is concretely expensive. Certainly, we need a new block encryption to reduce the overhead, which is our first challenge.

One might ask why not share plaintext blocks to enable 2PC for access and eviction to avoid expensive secret-shared decryption. We avoid this for efficiency. First, standard 2-server PIR is incompatible with secret shares because access would require expensive interactive secret-shared multiplication. Secure multiplication between two secret-shared values requires interaction, while multiplying a secret-shared value by a public value requires no interaction. If blocks were shared, it would require multiplication between the secret-shared one-hot vector and secret-shared blocks, incurring $O(Z \log N)$ blocks of inter-server communication per access. This motivates prior 3-server works [23, 24, 38] where redundancy enables efficient PIR, but this option is unavailable here. Second, eviction is infrequent. With parameters from [14, 24], the eviction frequency A exceeds 300. Therefore, a lightweight access protocol is desirable.

We propose a novel block encryption mechanism that allows the servers to non-interactively convert encrypted blocks to the secret-sharing of the encrypted plaintext blocks. This enables the servers to share the underlying plaintext blocks before eviction *for free*. Then we propose a new oblivious permutation protocol, with a new offline-online decomposition that minimizes the client-server communication. We also propose a bundle of new tricks and optimizations that improve both asymptotic and concrete efficiency.

3.2 Encryption with Share Conversion

We propose a simple PRF-based encryption scheme. Our encryption is inspired by [15], yet we randomly choose nonces as the PRF inputs. The encryption scheme $\Pi_{\mathrm{SE}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Convert})$ is defined as in Algorithm 1. Notably, this scheme has an additional conversion algorithm that converts any encrypted ciphertext to a secret sharing of the encrypted plaintext. Note that in the following sections, we may use the proposed symmetric encryption implicitly.

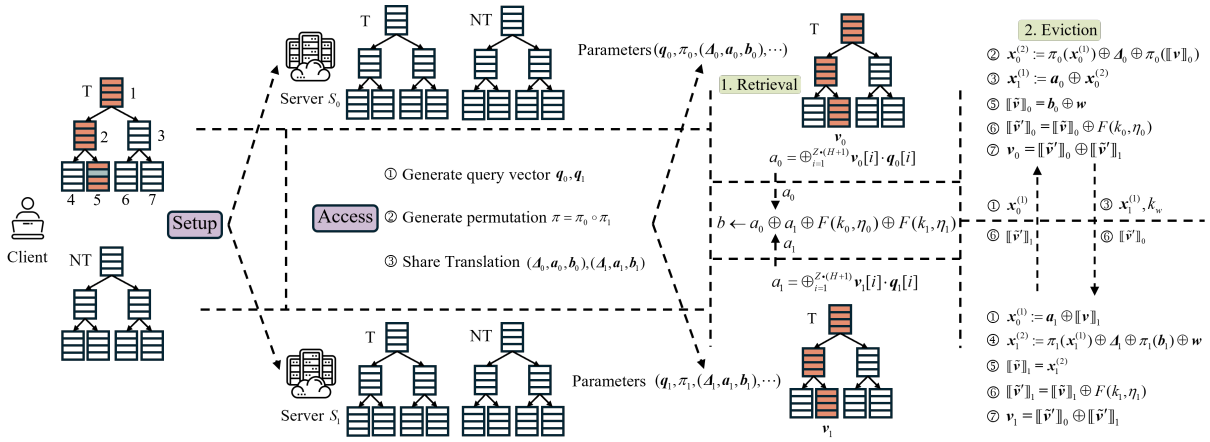


Figure 2: Overview of the DuetORAM System Workflow

Algorithm 1: The Symmetric Encryption Scheme

Parameters: PRF $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ with $\mathcal{K} = \{0,1\}^\lambda$, $\mathcal{X} = \{0,1\}^\lambda$, $\mathcal{Y} = \{0,1\}^B$.

$\mathsf{Gen}(1^\lambda)$:

Sample $k_0, k_1 \xleftarrow{\$} \{0,1\}^\lambda$;
return (k_0, k_1).

$\mathsf{Enc}(k_0, k_1, m)$:

Samples $\eta_0, \eta_1 \xleftarrow{\$} \{0,1\}^\lambda$;
Compute $c = F(k_0, \eta_0) \oplus F(k_1, \eta_1) \oplus m$;
return (η_0, η_1, c).

$\mathsf{Dec}(k_0, k_1, (\eta_0, \eta_1, c))$:

Compute $m = F(k_0, \eta_0) \oplus F(k_1, \eta_1) \oplus c$;
return m.

`// Convert a ciphertext to a share`

$\mathsf{Convert}(t, k_t, (\eta_0, \eta_1, c))$: ▷ $t \in \{0,1\}$

Compute $m_t = F(k_t, \eta_t) \oplus t \cdot c$;
return m_t.

3.3 DuetORAM Setup Protocol

Protocol 1 shows the DuetORAM setup procedure. The client uses the proposed encryption scheme to encrypt ORAM blocks. At the end of the setup, the client sends ciphertexts, nonces, and key k_t to server S_t.

Specifically, the client initializes two complete binary trees T and NT of height H. Here, T stores the encrypted data blocks, while NT stores the corresponding *nonce*s (ηs). Specifically, the trees share an identical structure such that for every block at $\mathsf{T}[i, j]$, the associated ηs are stored at $\mathsf{NT}[i, j]$.

Under this parameterization, the tree stores at most $N \leq A \cdot 2^{H-1}$ data blocks, where A denotes the eviction frequency. Each node holds up to Z blocks and is initialized with Z dummy blocks represented as B-bit zero strings. Two counters, n_r and n_e, are initialized to track the cumulative number

Protocol 1: $\mathsf{DuetORAM.Setup}(1^\lambda, \mathsf{DB})$

@Client:

Organize DB into blocks $(b_1, \cdots, b_N)$ with IDs $(\mathsf{id}_1, \cdots, \mathsf{id}_N)$;
Initialize two complete binary trees T and NT of height H, where $\mathsf{T}[i, j] \leftarrow \{0\}^B$ and $\mathsf{NT}[i, j] \leftarrow \{0\}^\lambda$ for all $i \in [2^{H+1} - 1]$ and $j \in [Z]$;
$n_r, n_e \leftarrow 0$;
for $i \in \{1, \cdots, N\}$ **do**
  $z_i \xleftarrow{\$} [2^H, 2^{H+1})$;
  Select (x_i, y_i), s.t. $x_i \in \mathcal{P}(z_i), y_i \in [Z]$, and $\mathsf{T}[x_i, y_i]$ is empty;
  $\mathsf{T}[x_i, y_i] \leftarrow b_i$, $\mathsf{md}[x_i][y_i] \leftarrow \mathsf{id}_i$;
  $\mathsf{pm}[\mathsf{id}_i] \leftarrow (z_i, \lfloor \log_2 x_i \rfloor \cdot Z + y_i)$;
$(k_0, k_1) \leftarrow \Pi_{\mathrm{SE}}.\mathsf{Gen}(1^\lambda)$;
for $i \in \{1, \cdots, 2^{H+1} - 1\}$ **do**
  for $j \in \{1, \cdots, Z\}$ **do**
    $(\eta_{0,i,j}, \eta_{1,i,j}, c_{i,j}) \leftarrow \Pi_{\mathrm{SE}}.\mathsf{Enc}(k_0, k_1, b_{i,j})$;
    $\mathsf{NT}[i, j] \leftarrow (\eta_{0,i,j}, \eta_{1,i,j})$, $\mathsf{T}[i, j] \leftarrow c_{i,j}$;
return $(\mathsf{T}, \mathsf{NT}, k_t)$. ▷ Distribute $(\mathsf{T}, \mathsf{NT}, k_t)$ to server S_t, for $t \in \{0,1\}$.

of retrievals and evictions, respectively. Blocks from the input database DB are organized with IDs from id_1 to id_N and distributed randomly among the leaf nodes of the tree, with their locations recorded accordingly in the position map pm and bucket metadata md. Following standard tree-based ORAM paradigms [24, 32], storing pm and md locally on the client incurs a linear client storage overhead of $O(N \cdot \log N)$ bits. Since pm and md only track structural routing and block identifiers rather than the data payloads themselves, this local memory footprint remains negligible relative to the server-side outsourced storage $O(N \cdot B)$, particularly in practical scenarios where the block size $B \gg \log N$.

3.4 DuetORAM Access Protocol

Upon receiving an access request, the DuetORAM access protocol executes an oblivious retrieval operation, and triggers the eviction procedure periodically based on the access counter. The eviction operation relies on several subroutines, which we elaborate below. Protocol 2 presents the complete DuetORAM.Access procedure.

Protocol 2: $b \leftarrow$ DuetORAM.Access(op, id, b^*)

$b \leftarrow$ DuetORAM.Retrieve(id);
$z \xleftarrow{\$} [2^H, 2^{H+1})$;
pm[id] $\leftarrow (z, n_r + 1)$;
if op = read **then**
  $b^* \leftarrow b$;
$(\eta_0, \eta_1, c^*) \leftarrow \Pi_{\mathrm{SE}}.\mathsf{Enc}(k_0, k_1, b^*)$;
Write c^* to T$[1, n_r + 1]$ and (η_0, η_1) to NT$[1, n_r + 1]$;
$n_r \leftarrow n_r + 1 \bmod A$, md$[1][n_r] \leftarrow$ id;
if $n_r = 0$ **then**
  Execute DuetORAM.Eviction protocol;
  $n_e \leftarrow n_e + 1 \bmod 2^H$;
return b.

3.4.1 Retrieval

Recall that in DuetORAM, both servers maintain identical copies of the encrypted database. These copies are masked by the outputs of pseudo-random functions under keys k_0 and k_1, which are held exclusively by S_0 and S_1, respectively. This architecture enables oblivious retrieval employing a classical 2-server PIR-based approach [11].

To retrieve a target block id, the client first identifies its logical location within the tree using the position map pm. The client then generates queries $\boldsymbol{q}_0$ and $\boldsymbol{q}_1$ of length $n = Z \cdot (H+1)$, representing the total number of blocks along the access path. Upon receiving the query, each server constructs a path vector $\boldsymbol{v}_t$ by concatenating the blocks along the requested access path. The server then invokes PIR.Retrieve$(t, \boldsymbol{q}_t, \boldsymbol{v}_t)$[3]. Finally, the client recovers the target block b by aggregating the responses. Protocol 3 provides the details of the retrieval.

3.4.2 Eviction

After every A accesses, DuetORAM deterministically selects a path based on the reverse lexicographical order [18] and performs eviction along it. In our binary ORAM tree, edges to left children are labeled as 0 and those to right children as 1. The eviction path corresponding to the n_e-th eviction is the bit-reversed binary encoding of n_e.

[3]This retrieval yields shares of both the target block and the associated nonces (η_1, η_2). For brevity, subsequent descriptions focus on block reconstruction; nonce reconstruction proceeds identically and is thus omitted.

Protocol 3: $b \leftarrow$ DuetORAM.Retrieve(id)

@Client:
(pID, pIdx) $\leftarrow$ pm[id];
$(\boldsymbol{q}_0, \boldsymbol{q}_1) \leftarrow$ PIR.Create(pIdx);
Send $(\mathsf{pID}, \boldsymbol{q}_t)$ to server S_t for $t \in \{0, 1\}$.

@Server: each server S_t receiving $(\mathsf{pID}, \boldsymbol{q}_t)$, **do**
Construct path vector $\boldsymbol{v}_t$ by concatenating blocks along path pID on server S_t, ordered from root to leaf;
$[\![c]\!]_t \leftarrow$ PIR.Retrieve$(t, \boldsymbol{q}_t, \boldsymbol{v}_t)$;
Send $[\![c]\!]_t$ to the client;

@Client: on receiving the shares $[\![c]\!]_0$ and $[\![c]\!]_1$, **do**
$c \leftarrow$ PIR.Reconstruct$([\![c]\!]_0, [\![c]\!]_1)$;
$b \leftarrow c \oplus F(k_0, \eta_0) \oplus F(k_1, \eta_1)$;
return b.

The core idea of the triplet-eviction strategy is to move each real block in the bucket $\mathcal{P}(p_e, h)$ on path p_e to one of its two child buckets $\mathcal{P}(p_e, h+1)$ or its sibling, for each level $h \in [0, H)$. We refer to $\mathcal{P}(p_e, h)$ as the *source bucket*, $\mathcal{P}(p_e, h+1)$ as the *destination bucket*, and the other child as the *sibling bucket* (see Figure 3 for clarification). As required by the ORAM correctness, each block must remain on the path to its mapped leaf after eviction.

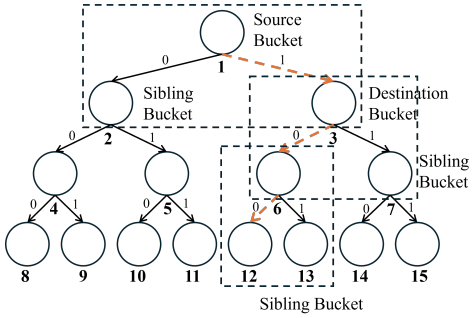


Figure 3: Triplet Eviction Clarification.

Based on the bucket overflow probability (Lemma 1) and the definition of reverse lexicographical order [18], a structural property of triplet eviction yields the following observation. Immediately prior to each eviction, the sibling bucket of every internal node along the eviction path is empty (excluding the leaf level). After the eviction completes at a given level, the source bucket becomes empty. Consequently, blocks whose leaf labels lie on the eviction path are eventually pushed all the way down to the leaf bucket, whereas blocks whose labels do not lie on the path are diverted into the sibling bucket.

The above insight implies that the cumulative effect of the H sequential triplet evictions is equivalent to a single batch operation, and the final position of each block depends solely on its assigned leaf, independent of the intermediate steps. Therefore, instead of evicting level by level, we treat all blocks on the eviction path together with those in the leaf-level sibling bucket as a single pool, apply one oblivious permutation

that places each block in the deepest bucket consistent with its leaf label, and then materialize the resulting bucket contents. For example, in Figure 3, after shuffling, the blocks remaining in bucket 12 are unchanged, bucket 13 is replaced by the contents of bucket 6, bucket 7 is replaced by bucket 3, and bucket 2 is replaced by bucket 1.

To facilitate a secret-shared permutation of blocks along the eviction path, the system must first transform the stored double-masked encrypted blocks into additive XOR shares. Practically, this requires one server to remove its corresponding PRF masking layer from the stored data, yielding a share masked solely by the peer's key. The peer server, in turn, utilizes its locally generated PRF mask as the complementary share. Through this procedure, the two servers effectively establish valid XOR shares of the underlying data blocks *without interaction*.

With these shares established non-interactively, the remaining core task of eviction reduces to a secure data-dependent permutation. To abstract away the low-level cryptographic mechanics and present a clean top-down protocol layout, we formalize this step using an ideal path permutation functionality, denoted as $\mathcal{F}_{\mathsf{perm}}$ (Functionality 1). Formally, $\mathcal{F}_{\mathsf{perm}}$ takes the target permutation π from the client, the plaintext block shares $[\![\boldsymbol{v}]\!]_0$ and $[\![\boldsymbol{v}]\!]_1$ from the two servers, and outputs the permuted shares $[\![\tilde{\boldsymbol{v}}]\!]_0$ and $[\![\tilde{\boldsymbol{v}}]\!]_1$ such that $[\![\tilde{\boldsymbol{v}}]\!]_0 \oplus [\![\tilde{\boldsymbol{v}}]\!]_1 = \pi(\boldsymbol{v})$. We provide the detailed definitions of correctness and security in Appendix §C.3.

Functionality 1: Ideal Secret-shared Permutation Functionality $\mathcal{F}_{\mathsf{perm}}$

Parameters: A permutation $\pi \in \mathbf{S}_n$ from the client and a plaintext share vector $[\![\boldsymbol{v}]\!]_t \in (\{0,1\}^B)^n$ from each server S_t, $t \in \{0,1\}$.

Functionality: After receiving a permutation π from the client, a share vector $[\![\boldsymbol{v}]\!]_0$ from server S_0, and a share vector $[\![\boldsymbol{v}]\!]_1$ from server S_1:

1 Compute $\boldsymbol{v} = [\![\boldsymbol{v}]\!]_0 \oplus [\![\boldsymbol{v}]\!]_1$ and $\tilde{\boldsymbol{v}} = \pi(\boldsymbol{v})$;

2 Sample $\boldsymbol{r} \xleftarrow{\$} (\{0,1\}^B)^n$, and set $[\![\tilde{\boldsymbol{v}}]\!]_0 = \boldsymbol{r}$, $[\![\tilde{\boldsymbol{v}}]\!]_1 = \tilde{\boldsymbol{v}} \oplus \boldsymbol{r}$;

3 Send $[\![\tilde{\boldsymbol{v}}]\!]_t$ to server S_t for $t \in \{0,1\}$.

After the execution of the path permutation, a reverse transformation is required to convert the permuted XOR shares back into the standard double-masked format for storage. This reconstruction necessitates one additional round of interaction. Specifically, each server S_t generates a "blinded share" by XORing its local secret share with a fresh PRF mask derived from the new nonce. The servers then exchange these blinded shares. Finally, by XORing the locally computed blinded share with the received one, both servers reconstruct the identical double-masked encrypted blocks (i.e., $b \oplus F(k_0,\eta_0) \oplus F(k_1,\eta_1)$).

In summary, the eviction procedure proceeds as follows. The client computes the permutation π that maps blocks on the eviction path to their target destinations. The servers then non-interactively convert the encrypted blocks into secret shares, after which the parties invoke the ideal functionality $\mathcal{F}_{\mathsf{perm}}$ to securely permute the shares. Finally, the servers re-encrypt the permuted blocks and write them back to their corresponding buckets. Protocol 4 formally describes this eviction procedure in the $\mathcal{F}_{\mathsf{perm}}$-hybrid model, integrating the ideal functionality with its supporting subroutines. Below, we first detail these auxiliary subroutines, and then present the concrete protocols that instantiate $\mathcal{F}_{\mathsf{perm}}$ in Section 3.5.

Protocol 4: DuetORAM.Eviction(n_e)

@Client:

1 $p_e \leftarrow \mathsf{bitreverse}(n_e)$;

2 **for** $l \in \{0,\cdots,H\}$ **do**

3 $\quad S \leftarrow S \cup \mathsf{ReadBucket}(\mathcal{P}(p_e,l))$;

4 Let $\hat{u}_H$ be the index of the sibling bucket of $\mathcal{P}(p_e,H)$;

5 $S \leftarrow S \cup \mathsf{ReadSibling}(\hat{u}_H)$;

6 $\pi \leftarrow \mathsf{GeneratePermutation}(S, \mathsf{pID})$;

7 The client sends π to $\mathcal{F}_{\mathsf{perm}}$;

@Server:

1 Let $\boldsymbol{v}_t$ be the vector of data blocks along path p_e ordered from root to leaf, followed by the blocks in the bucket $\hat{u}_H$;

▷ The size of $\boldsymbol{v}_t$ is $Z \cdot (H+2)$.

2 Locally convert $\boldsymbol{v}_t$ into secret shares $[\![\boldsymbol{v}]\!]_t$;

3 S_t calls $\mathcal{F}_{\mathsf{perm}}$ with input $[\![\boldsymbol{v}]\!]_t$ and obtains shares $[\![\tilde{\boldsymbol{v}}]\!]_t$;

4 Generate nonce list $\mathcal{L}_{\eta_t} \leftarrow G(\mathsf{s}\eta_t)$, $\mathsf{s}\eta_t \xleftarrow{\$} \{0,1\}^\lambda$;

5 Compute $[\![\tilde{\boldsymbol{v}}']\!]_t[i] \leftarrow [\![\tilde{\boldsymbol{v}}]\!]_t[i] \oplus F(k_t,\eta_i)$ for all $i \in [Z \cdot (H+2)]$, where $\eta_i \leftarrow \mathcal{L}_{\eta_t}[i]$;

6 S_t sends $(\mathsf{s}\eta_t, [\![\tilde{\boldsymbol{v}}']\!]_t)$ to the other server S_{1-t};

7 Reconstruct the path data, $\boldsymbol{v}_t \leftarrow [\![\tilde{\boldsymbol{v}}']\!]_0 \oplus [\![\tilde{\boldsymbol{v}}']\!]_1$.

• ***Read Blocks***. Before constructing the eviction permutation, the client determines the exact locations of all real blocks on the path. Since the position map maintains each block's assigned leaf and the metadata tracks the occupancy of every bucket, the client can locally reconstruct the set of blocks residing on the path without interacting with either server.

Algorithm 5: Read Algorithms

// Read the metadata of bucket i.

$S \leftarrow \mathsf{ReadBucket}(i)$:

1 **for** $j \in \{1,\cdots,Z\}$ **do**

2 $\quad \mathsf{id}_j \leftarrow \mathsf{md}[i][j]$, $(\mathsf{pID},\mathsf{pIdx}) \leftarrow \mathsf{pm}[\mathsf{id}_j]$;

3 $\quad S \leftarrow S \cup (\mathsf{id}_j, \mathsf{pID}, \mathsf{pIdx})$;

4 **return** S.

$S \leftarrow \mathsf{ReadSibling}(i)$:

1 **for** $j \in \{1,\cdots,Z\}$ **do**

2 $\quad \mathsf{id}_j \leftarrow \mathsf{md}[i][j]$, $(\mathsf{pID},\mathsf{pIdx}) \leftarrow \mathsf{pm}[\mathsf{id}_j]$;

3 $\quad S \leftarrow S \cup (\mathsf{id}_j, \mathsf{pID}, \mathsf{pIdx}+Z)$;

4 **return** S.

To support the one-shot eviction optimization, we include the leaf-level sibling bucket in the shuffle domain. Conceptually, we treat this bucket as the $(H+1)$-th bucket on the eviction path. Accordingly, the logical path indices of blocks stored in this sibling bucket are offset by Z, ensuring that all blocks across the augmented path occupy a single contiguous index space. The procedure is outlined in Algorithm 5.

• ***Generate Permutation***. Once the locations of all real blocks on the eviction path have been determined, the client constructs a permutation that pushes these blocks as deep into the tree as possible. The permutation is derived by greedily filling buckets starting at the leaf and proceeding upward toward the root, ensuring that blocks occupy the deepest bucket consistent with their assigned leaf. Concretely, a block id' mapped to leaf pID' may be placed in a bucket at level l if and only if the two paths intersect at that level, i.e., $\mathcal{P}(\mathsf{pID}, l) = \mathcal{P}(\mathsf{pID}', l)$.

A structural property of triplet eviction implies that once eviction completes, all blocks residing in internal buckets along the eviction path will be relocated into the sibling bucket at the next depth, rather than remaining on the path itself. Consequently, the path index of each displaced block is shifted downward by Z positions. To prevent the server from correlating pre- and post-shuffle blocks via reuse of nonces, the nonces for all blocks involved in the shuffle are randomly refreshed. Algorithm 6 specifies the complete permutation construction procedure.

Algorithm 6: Generate Permutation

```
// Derive eviction permutation π
π ← GeneratePermutation(S, pID):
for l ∈ {H, H−1, ···, 0} do
    S' ← {(id', pID', pIdx') ∈ S : P(pID, l) = P(pID', l)};
    S' ← Select min(|S'|, Z) blocks from S';
    S ← S − S';
    if l = H then
        Clear md[P(pID, l)];
        for i ∈ {1, ···, |S'|} do
            π(Z * l + i) ← pIdx'_i;
            pm[id'_i] ← (pID'_i, Z · l + i);
            md[P(pID, l)][i] ← id'_i;
    else
        Set u ← P(pID, l), and let û be the index of the
          sibling of node P(pID, l + 1);
        Clear md[u] and md[û];
        for i ∈ {1, ···, |S'|} do
            π(Z · l + i) ← pIdx'_i;
            pm[id'_i] ← (pID'_i, Z · (l + 1) + i);
            md[û][i] ← id'_i;
Pad the remaining slots in π with dummy indices;
return π.
```

3.5 Instantiating $\mathcal{F}_{\mathsf{perm}}$: From Pure-Online to Offline-Online Decomposition

To realize the ideal functionality $\mathcal{F}_{\mathsf{perm}}$ with a trusted client and two non-colluding servers, we present two alternative protocol instantiations. We first describe a straightforward pure-online protocol inspired by the CGP shuffle, and then present our optimized offline-online protocol tailored for large ORAM blocks and bandwidth efficiency.

3.5.1 Pure-Online Instantiation

The ideal functionality $\mathcal{F}_{\mathsf{perm}}$ targets a secure permutation of the data vector shares $[\![\boldsymbol{v}]\!]_t$ across both servers based on a target permutation π, yielding the permuted shares $[\![\tilde{\boldsymbol{v}}]\!]_t$. Naturally, $\mathcal{F}_{\mathsf{perm}}$ can be instantiated through a two-party secret-shared shuffle protocol inspired by the CGP shuffle. The standard CGP shuffle relies on an interactive *Share Translation* protocol where parties must invoke OT and puncturable PRFs to generate shuffle correlations. However, running such protocols online over a large ORAM path introduces extensive communication and computational overhead between the servers. Benefiting from our architecture featuring a trusted client and two non-colluding servers, we can strategically delegate the generation of these shuffle correlations to the client. This architectural optimization avoids executing the heavy *Share Translation* protocol over inter-server channels during eviction. We specify the concrete shuffle procedure below.

• ***Secret Shared Shuffle.*** The client first takes the target permutation π and randomly decomposes it into two sub-permutations π_0 and π_1 such that $\pi = \pi_0 \circ \pi_1$. The client then generates the random vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ for each sub-permutation, as well as the translation vector $\boldsymbol{\Delta}$ required for the oblivious shuffle. Specifically, concerning the sub-permutation π_t ($t \in \{0,1\}$), the client randomly selects vectors $\boldsymbol{a}_{1-t}$ and $\boldsymbol{b}_{1-t}$ of size n_s, where n_s signifies the size of the data vector shares to be permuted, and computes $\boldsymbol{\Delta}_t$ as $\boldsymbol{\Delta}_t = \pi_t(\boldsymbol{a}_{1-t}) \oplus \boldsymbol{b}_{1-t}$. Subsequently, the client distributes the tuple $(\pi_t, \boldsymbol{\Delta}_t, \boldsymbol{a}_t, \boldsymbol{b}_t)$ to each respective server S_t. Additionally, let $H : \{0,1\}^* \to (\{0,1\}^B)^{n_s}$ be a hash function.

Based on the received parameters, both servers conduct the interactive protocol for the secret-shared shuffle. More concretely, S_1 sends its masked data $\boldsymbol{x}_0^{(1)} := \boldsymbol{a}_1 \oplus [\![\boldsymbol{v}]\!]_1$ to S_0. Upon receiving the masked secret share $\boldsymbol{x}_0^{(1)}$ from S_1, S_0 computes $\boldsymbol{x}_0^{(2)} := \pi_0(\boldsymbol{x}_0^{(1)}) \oplus \boldsymbol{\Delta}_0 \oplus \pi_0([\![\boldsymbol{v}]\!]_0)$ and forwards $\boldsymbol{x}_1^{(1)} := \boldsymbol{a}_0 \oplus \boldsymbol{x}_0^{(2)}$ along with a randomly sampled seed $k_w \xleftarrow{\$} \{0,1\}^\lambda$ to S_1. After receiving the information from S_0, S_1 locally computes the pseudorandom vector $\boldsymbol{w} = H(k_w)$ and $\boldsymbol{x}_1^{(2)} := \pi_1(\boldsymbol{x}_1^{(1)}) \oplus \boldsymbol{\Delta}_1 \oplus \pi_1(\boldsymbol{b}_1) \oplus \boldsymbol{w}$. Ultimately, S_0 acquires the shuffled secret share $[\![\tilde{\boldsymbol{v}}]\!]_0 = \boldsymbol{b}_0 \oplus \boldsymbol{w}$, and S_1 obtains the shuffled secret share $[\![\tilde{\boldsymbol{v}}]\!]_1 = \boldsymbol{x}_1^{(2)}$, satisfying $[\![\tilde{\boldsymbol{v}}]\!]_0 \oplus [\![\tilde{\boldsymbol{v}}]\!]_1 = \pi_1(\pi_0([\![\boldsymbol{v}]\!]_0 \oplus [\![\boldsymbol{v}]\!]_1)) = \pi(\boldsymbol{v})$. Subroutine 1 formalizes these client operations and inter-server interactions.

Intuitively, the correctness and security of $\mathcal{F}_{\mathsf{perm}}$ inherit directly from the CGP shuffle. Since we merely offload the Share Translation to the client without altering the underlying cryptographic structure, these guarantees hold implicitly. Protocol 5 presents the pure-online instantiation of the functionality $\mathcal{F}_{\mathsf{perm}}$.

Subroutine 1: Secret Shared Shuffle based Two-Party Permutation Protocol ($\mathsf{S^3 2P^2}$)

// Generate shuffle correlations.
$((\pi_0, \Delta_0, \boldsymbol{a}_0, \boldsymbol{b}_0), (\pi_1, \Delta_1, \boldsymbol{a}_1, \boldsymbol{b}_1)) \leftarrow \mathsf{S^3 2P^2.ST}(\pi)$:

Sample random permutations π_0, π_1, s.t. $\pi = \pi_0 \circ \pi_1$;
Randomly select vectors of size $n_s, \boldsymbol{a}_0, \boldsymbol{b}_0, \boldsymbol{a}_1, \boldsymbol{b}_1$, where each element of the vector $\varepsilon \xleftarrow{\$} \{0,1\}^B$;
$\Delta_0 := \pi_0(\boldsymbol{a}_1) \oplus \boldsymbol{b}_1$;
$\Delta_1 := \pi_1(\boldsymbol{a}_0) \oplus \boldsymbol{b}_0$;
return $((\pi_0, \Delta_0, \boldsymbol{a}_0, \boldsymbol{b}_0), (\pi_1, \Delta_1, \boldsymbol{a}_1, \boldsymbol{b}_1))$.

// Two servers secret shared shuffle.
$\mathsf{S^3 2P^2.SecretShare}(S_0, S_1)$:

S_1 sends $\boldsymbol{x}_0^{(1)} := \boldsymbol{a}_1 \oplus [\![\boldsymbol{v}]\!]_1$ to S_0;
S_0 computes $\boldsymbol{x}_0^{(2)} := \pi_0(\boldsymbol{x}_0^{(1)}) \oplus \Delta_0 \oplus \pi_0([\![\boldsymbol{v}]\!]_0)$;
S_0 sends $\boldsymbol{x}_1^{(1)} := \boldsymbol{a}_0 \oplus \boldsymbol{x}_0^{(2)}$ and k_w to S_1, where $k_w \xleftarrow{\$} \{0,1\}^\lambda$, and locally computes $\boldsymbol{w} = H(k_w)$;
S_1 computes $\boldsymbol{x}_1^{(2)} := \pi_1(\boldsymbol{x}_1^{(1)}) \oplus \Delta_1 \oplus \pi_1(\boldsymbol{b}_1) \oplus \boldsymbol{w}$, where $\boldsymbol{w} = H(k_w)$;
S_0 outputs $[\![\tilde{\boldsymbol{v}}]\!]_0 = \boldsymbol{b}_0 \oplus \boldsymbol{w}$. S_1 outputs $[\![\tilde{\boldsymbol{v}}]\!]_1 = \boldsymbol{x}_1^{(2)}$.

Protocol 5: Pure-online Instantiation $\Pi_{\mathsf{perm\text{-}on}}$

@Client:
$((\pi_0, \Delta_0, \boldsymbol{a}_0, \boldsymbol{b}_0), (\pi_1, \Delta_1, \boldsymbol{a}_1, \boldsymbol{b}_1)) \leftarrow \mathsf{S^3 2P^2.ST}(\pi)$.

@Server:
$\mathsf{S^3 2P^2.SecretShare}(S_0, S_1)$.

3.5.2 Optimized Offline-Online Instantiation

Although the pure-online protocol (Section 3.5.1) significantly mitigates inter-server overhead by offloading the Share Translation to the client, it still requires the client to generate and transmit the full-size shuffle correlation vectors $(\Delta_t, \boldsymbol{a}_t, \boldsymbol{b}_t)$ during the eviction phase. Since the elements of these masking vectors must match the full ORAM block size B, thereby incurring $3n_sB$ bits of communication overhead per server, this design introduces a noticeable bandwidth bottleneck between the client and the servers when handling large block sizes. To overcome this limitation, we present an optimized offline-online instantiation.

Specifically, we decompose Share Translation into offline and online phases. In the offline phase, the client expands seeds k_3, k_4 into full $n_s \times n_s$ matrices $\boldsymbol{A}, \boldsymbol{B}$ via a PRG, and generates random punctured vectors $\boldsymbol{u}_0, \boldsymbol{u}_1$ of length n_s. Each element $\boldsymbol{u}_t[i]$ determines a position in the corresponding matrix to be punctured. For example, $\boldsymbol{u}_0[i]$ refers to element $\boldsymbol{A}[i][\boldsymbol{u}_0[i]]$ in matrix $\boldsymbol{A}$, which is set to zero. Once the punctured matrices $\boldsymbol{A}'$ and $\boldsymbol{B}'$ are obtained, the client can distribute $(k_4, \boldsymbol{A}')$ and $(k_3, \boldsymbol{B}')$ to servers S_0 and S_1, respectively, during idle periods (Subroutine 2). Note that these random punctures do not yet reflect the target permutation.

Subroutine 2: Secret Shared Shuffle based Two-Party Permutation Protocol - Offline($\mathsf{S^3 2P^2 O}$)

// Generate random permutation matrices.
$((\boldsymbol{A}', k_4), (\boldsymbol{B}', k_3), \boldsymbol{u}_0, \boldsymbol{u}_1) \leftarrow \mathsf{S^3 2P^2 O.Initialize}(n_s)$:

$k_3, k_4 \xleftarrow{\$} \{0,1\}^\lambda$;
Generate two $n_s \times n_s$ random matrices, $\boldsymbol{A}, \boldsymbol{B}$;
Randomly select two vectors of size n_s, $\boldsymbol{u}_0, \boldsymbol{u}_1$, each element in vectors $\varepsilon \xleftarrow{\$} \{1, \cdots, n_s\}$;
for $i \in \{1, \cdots, n_s\}$ **do**
  $\boldsymbol{A}'[i] \leftarrow \boldsymbol{A}[i]$, where $\boldsymbol{A}'[i][\boldsymbol{u}_0[i]] \leftarrow 0$;
  $\boldsymbol{B}'[i] \leftarrow \boldsymbol{B}[i]$, where $\boldsymbol{B}'[i][\boldsymbol{u}_1[i]] \leftarrow 0$;
Distribute $(\boldsymbol{A}', k_4)$ to S_0, and $(\boldsymbol{B}', k_3)$ to S_1.
▷ Based on the received parameters, the server S_t can compute $(\Delta_t, \boldsymbol{a}_t, \boldsymbol{b}_t)$.

In the online phase, we adapt the offline random correlations to the data-dependent target permutations. Upon decomposing the target permutation into sub-permutations π_0 and π_1, the client computes the cyclic shift vectors $\boldsymbol{r}_t[i] = (\pi_t[i] - \boldsymbol{u}_t[i]) \pmod{n_s}$ for $t \in \{0,1\}$ and distributes them to the servers. Using these cyclic shift vectors, the servers perform row-wise cyclic shifts on the full matrix (locally expanded from the PRG key) and the punctured matrix provided by the client (see Figure 4 for clarification). After obtaining the target matrices, they locally generate the corresponding pseudorandom shuffle correlations $(\Delta_t, \boldsymbol{a}_t, \boldsymbol{b}_t)$. Consequently, the per-server correlation overhead drops from $3n_sB$ bits in the pure-online variant to just $2n_s \log n_s$ bits, completely decoupling the transmission from the ORAM block size B.

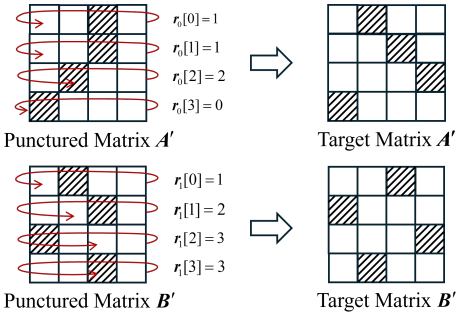


Figure 4: Row-wise Right Cyclic Shift

To further optimize bandwidth, we adapt the compact correlation design from [8] by having the client generate matrix elements as compact cryptographic seeds rather than full-size

blocks. The servers subsequently use a PRG to *locally stretch* these seeds to the ORAM block size B, which is required to mask the data and produce valid shares. This optimization effectively decouples the offline communication cost from the block size, yielding substantial savings for large blocks. Specifically, the offline communication overhead per server amounts to only $\lambda + n_s^2\lambda$ bits.

This data-independent structure partitions eviction into offline and online phases (Figure 5). Since pre-distributed matrices do not rely on runtime data, the client can proactively send multiple (seed, punctured matrix) pairs to the servers during idle periods to support consecutive accesses. Intuitively, the correctness and security of this offline-online instantiation hold implicitly. Because the puncture vectors $\boldsymbol{u}_t$ are uniformly random, the resulting cyclic shift vectors $\boldsymbol{r}_t$ are computationally masked and leak no target permutation information to an adversary. Since subsequent execution aligns with the CGP shuffle, all cryptographic guarantees remain intact. Protocol 6 presents the offline-online instantiation of the functionality $\mathcal{F}_{\mathsf{perm}}$.

Protocol 6: Offline-online Instantiation $\Pi_{\mathsf{perm\text{-}off}}$

@Client:

`// Offline phase.`

$((\boldsymbol{A}', k_4), (\boldsymbol{B}', k_3), \boldsymbol{u}_0, \boldsymbol{u}_1) \leftarrow \mathsf{S^3 2P^2 O.Initialize}(n_s)$;

`// Online phase.`

Sample random permutations π_0, π_1, s.t. $\pi = \pi_0 \circ \pi_1$;

for $i \in \{0, 1, \cdots, n_s - 1\}$ **do**

  $\boldsymbol{r}_t[i] \leftarrow (\pi_t[i] - \boldsymbol{u}_t[i]) \bmod n_s$;

Send $(\pi_t, \boldsymbol{r}_0, \boldsymbol{r}_1)$ to S_t, for $t \in \{0, 1\}$.

@Server:

`// Offline phase.`

Recover and expand the random matrix $\boldsymbol{B}$ (or $\boldsymbol{A}$) and the punctured matrix $\boldsymbol{A}'$ (or $\boldsymbol{B}'$);

`// Online phase.`

Cyclically shift the rows of matrices $\boldsymbol{A}', \boldsymbol{B}$ or $\boldsymbol{A}, \boldsymbol{B}'$ based on $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$, respectively;

Compute the share translations $(\Delta_t, \boldsymbol{a}_t, \boldsymbol{b}_t)$;

$\mathsf{S^3 2P^2.SecretShare}(S_0, S_1)$.

3.6 Theoretical Cost Analysis

We provide a theoretical analysis of the communication and computation costs of DuetORAM during the retrieval and eviction phases, and quantify the corresponding overhead incurred by the client and the servers.

• ***Retrieval***. At the beginning of the retrieval phase, the client invokes PIR.Create to generate a binary query vector of dimension $Z \cdot (H+1)$, incurring $Z \cdot (H+1)$ local XOR operations. Crucially, the query size depends only on the number of blocks along the access path and remains independent of the block size. Thus, the client-server communication during query issuance consists of a $Z \cdot (H+1)$-bit vector together with the path identifier.

Upon receiving the query, each server computes a masked share of the requested block by evaluating the XOR-reduced product $[\![c]\!]_t = \bigoplus_i^n \boldsymbol{q}_t[i] \cdot \boldsymbol{v}_t[i]$, where $\boldsymbol{q}_t[i] \in \{0,1\}$ selects entries along the access path, the multiplication is ordinary multiplication, and the reduction is taken under bitwise XOR. In addition, each server computes the shares of the two associated nonces. Each server returns its shares to the client, which reconstructs the block and subsequently transmits a fresh double-masked encrypted block along with two new nonces back to the servers during write-back.

Overall, the client-server communication complexity of a single retrieval is $O(Z \cdot H + B)$, where B is the block size, and the computational complexity is $O(Z \cdot H)$.

• ***Eviction***. In the offline phase, the client sends to each server a matrix-key pair consisting of an $n_s \times n_s$ matrix and a secret key, resulting in $n_s^2 \cdot \lambda + \lambda$ bits of client-server communication. In the online phase, the client supplies a sub-permutation and two cyclic shift vectors. The total client-server online communication is $3 \cdot n_s \cdot \log n_s$ bits.

Upon receiving these values, each server applies cyclic shifts to the received matrices and computes the corresponding share translations on $2n_s^2$ elements. Servers then jointly shuffle along the eviction path and exchange blinded shares to derive consistent double-masked encrypted blocks. This step requires $4n_s$ operations per server and $2 \cdot Z \cdot B \cdot (H+2) + \lambda$ bits of inter-server communication.

Overall, for a single eviction, the offline client-server communication complexity is $O(\lambda \cdot Z^2 \cdot H^2)$, the online client-server communication complexity is $O(Z \cdot H \cdot \log Z \cdot H)$, the inter-server communication complexity is $O(Z \cdot B \cdot H)$, and the server-side computation complexity is $O(Z^2 \cdot H^2)$.

• ***Recursion Technique***. The standard ORAM recursion technique [34, 37] can be optionally employed to further reduce the client-side storage overhead. By setting the block size of the position map ORAMs to $\chi \log N$ ($\chi \geq 2$), each block stores χ position map entries from the preceding level. After $O\left(\frac{\log N}{\log \chi}\right)$ recursion levels, the final position map has constant size. This recursive structure increases the total inter-server communication complexity during eviction to $\sum_{k=0}^{\frac{\log N}{\log \chi}} O\left(\log \frac{N}{\chi^k}\right) = O\left(\frac{\log^2 N}{\log \chi}\right)$ blocks and scales the retrieval round complexity to $O\left(\frac{\log N}{\log \chi}\right)$. In this paper, we assume the client stores the position map locally without recursion (Section 3.3), under which our claimed $O(\log N)$ block communication complexity strictly holds.

4 Security Analysis

Lemma 1 (Overflow Probability [14]). *If $Z \geq A$ and $N \leq A \cdot 2^{H-1}$, the probability a bucket overflows after an eviction*

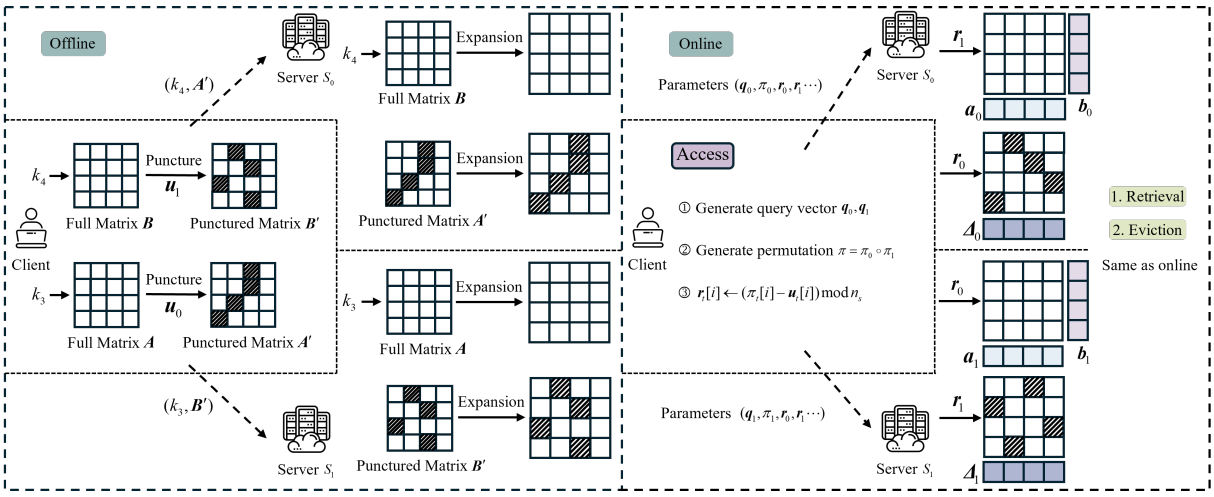


Figure 5: Overview of the DuetORAM Online and Offline Workflow. **Expansion**: each matrix entry is a compact seed that is locally expanded by the server into a block-sized value using a PRG.

operation is bounded by $e^{-\frac{(2Z-A)^2}{6A}}$*, where* $Z = A = \Theta(\lambda)$.

Proof. See [14] for a detailed proof. □

Theorem 1. DuetORAM *is a secure two-server DORAM construction in the* $\mathcal{F}_{\text{perm}}$*-hybrid model if* F *is a secure PRF,* G *is a secure PRG, and the underlying PIR protocol is secure (see Appendix §B for the formal PIR security definition).*

Proof. See Appendix §D.2. □

5 Evaluation

We evaluate the performance of DuetORAM through a prototype-based experimental study. We first describe our implementation details and experimental configuration. We then measure the performance of DuetORAM under different network conditions and varying database sizes. Finally, under a uniform setting where all schemes maintain the position map locally on the client without recursion, we compare DuetORAM with representative prior works, including the 2-server DUORAM [38] and the 3-server S^{3}ORAM [24], to demonstrate the efficiency of our design.

5.1 Implementation Experimental Setup

We implement a prototype of DuetORAM in C++, relying on two libraries: (i) the emp-tool library[4], which provides primitives for pseudorandom generation, pseudorandom function, and AES encryption in counter mode utilizing AES-NI hardware instructions; and (ii) ZeroMQ[5], which is used to support efficient communication between the client and servers.

[4]Available at https://github.com/emp-toolkit/emp-tool
[5]Available at https://zeromq.org/download/

Experiments are conducted on a machine with a 13th Gen Intel Core i7-1360P CPU @2.20GHz, 32GB RAM, and Ubuntu 22.04. We deploy one logical client and multiple logical servers on the same physical machine, using Linux `tc` to emulate network conditions for controlled evaluation. We consider two settings: a local-area network (LAN) with 30 μs round-trip time (RTT) and 100 Gbps bandwidth, and a wide-area network (WAN) with 30 ms RTT and 100 Mbps bandwidth. The security parameter is $\lambda = 128$. Following [24], we set the bucket size $Z = 333$ and eviction frequency $A = 280$ for consistency, though both are tunable.

We evaluate DuetORAM and its counterparts using randomly generated databases, measuring how database and block sizes affect retrieval latency, eviction performance, and communication overhead under both network settings.

We select DUORAM and S^{3}ORAM as baselines. DUORAM is a recent 2-server DORAM scheme aligned with our setting, while S^{3}ORAM provides efficient retrieval and eviction but requires at least three servers. These schemes represent the most relevant high-performance designs in the 2-server and multi-server DORAM settings. Other DORAM constructions [14, 15, 33, 37] are excluded, as their performance is already captured by DUORAM or S^{3}ORAM.

5.2 Main Results

We first evaluate the impact of database size and block size on retrieval performance under different network settings. We then compare DuetORAM with S^{3}ORAM in eviction overhead and communication cost. DUORAM is treated differently, as it performs an update after each access to maintain obliviousness and thus requires no separate eviction or shuffle. We therefore report only DUORAM's online retrieval cost. In the figures, the suffixes Off and On indicate whether DuetORAM

uses preprocessing during eviction, while 64 bytes and 128 bytes denote different block size configurations. Detailed raw data for all results in this section, covering both LAN and WAN settings and all scheme variants, are provided in Appendix E (Tables 3–8).

Retrieval Performance. Figure 6 compares the retrieval latency of DuetORAM and its counterparts under different network settings and parameters. Since retrieval involves no offline preprocessing, we do not distinguish between the two variants of DuetORAM (pure-online and offline-online) in this comparison, as all reported latency metrics reflect the retrieval time for both variants. Figures 6(a) and (b) show the retrieval latency with 8-byte blocks. Under LAN (Figure 6(a)), DuetORAM achieves substantially lower latency than DUORAM and S^{3}ORAM across all database sizes. For example, at 2^{17} blocks, DuetORAM takes about 1.1 ms per access, roughly 170× faster than DUORAM and 1.7× faster than S^{3}ORAM. As the database grows, the gap between DuetORAM and DUORAM widens further, highlighting our design's scalability. This improvement is primarily *computation-driven*: DuetORAM achieves logarithmic complexity using lightweight XOR operations, whereas DUORAM requires a linear scan over the entire database and S^{3}ORAM relies on heavier modular arithmetic. Under WAN (Figure 6(b)), the performance gap between DuetORAM and S^{3}ORAM narrows, as network latency dominates the end-to-end latency. Nevertheless, DuetORAM still outperforms DUORAM by a large margin across all database sizes.

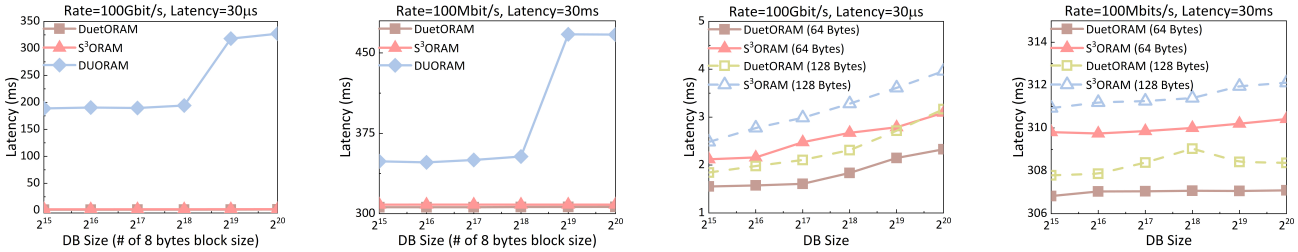
Figures 6(c) and (d) further evaluate retrieval performance with larger block sizes (64 and 128 bytes). In the LAN setting (Figure 6(c)), DuetORAM maintains a clear advantage over S^{3}ORAM for both block sizes, and the gap widens slightly as the database grows. In the WAN setting (Figure 6(d)), the curves flatten for all schemes due to dominant network delay, but DuetORAM still achieves the lowest or comparable latency across configurations. This advantage mainly stems from DuetORAM's use of lightweight XOR-based operations in the online phase, whereas S^{3}ORAM relies on more expensive modular arithmetic, leading to higher computation overhead, especially in low-latency networks.

Eviction Performance. Figure 8 compares the eviction latency of DuetORAM and S^{3}ORAM under different network settings. We evaluate two variants of DuetORAM: DuetORAM Off, which uses offline preprocessing, and DuetORAM On, which performs eviction fully online. As shown in Figure 8(a), under LAN setting, DuetORAM On consistently outperforms both DuetORAM Off and S^{3}ORAM across all configurations. For example, with a 64-byte block size and a database size of 2^{17}, DuetORAM On completes an eviction in 0.28 s, which is approximately 7× faster than S^{3}ORAM (1.90 s) and 4× faster than DuetORAM Off (1.19 s). This prominent advantage stems from DuetORAM On's single-shot whole-path eviction strategy and its lightweight, XOR-based data movement. In contrast, S^{3}ORAM demands multi-round server interactions and secure multiparty multiplication for path eviction, while DuetORAM Off incurs additional local computational overhead due to row-wise cyclic shifts and matrix expansions. Additionally, DuetORAM Off exhibits a notable block-size dependency in low-latency networks. As the block size increases to 128 bytes under the LAN setting, the local computational overhead of DuetORAM Off, specifically from matrix expansions, scales up significantly. Under WAN conditions (Figure 8(b)), the advantages of both DuetORAM variants become even more pronounced. Since wide-area latency severely amplifies communication overhead, protocols with higher interaction complexity incur significantly larger delays.

Furthermore, it can be observed from Figure 8 that DuetORAM On consistently yields lower latency than DuetORAM Off across all configurations, which perfectly aligns with our design expectations. Because the preprocessed variant (DuetORAM Off) is specifically engineered to optimize online client-server communication, it inherently introduces an alternative computation-driven bottleneck during the online phase. Concretely, DuetORAM Off requires the servers to perform local PRG expansions and row-wise cyclic shifts over $Z^2(H+2)^2$ elements to dynamically derive the expected shuffle correlations. Conversely, in DuetORAM On, these correlations are entirely computed by the client and transmitted directly to the servers, thereby eliminating the server-side local data reorganization delay at runtime.

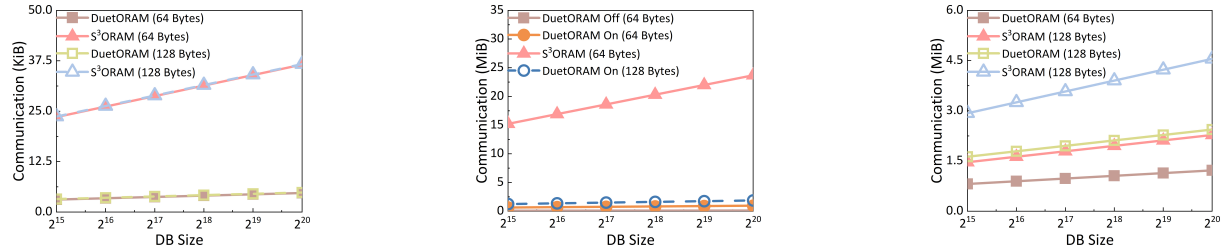
Communication Performance. Figure 7 compares the communication overhead of DuetORAM during retrieval and eviction. Figure 7(a) reports the client-server communication per restrieval. With 64-byte blocks and a database size of 2^{17}, DuetORAM (pure-online and offline-online) incurs about 3.8 KiB, approximately 87% lower than the 28.8 KiB required by S^{3}ORAM. This reduction stems from the underlying retrieval primitives: S^{3}ORAM uses Shamir secret sharing over $\mathbb{F}_p$, where each query entry is a field element, while DuetORAM adopts XOR-based PIR with bit-level query entries, significantly reducing communication. For both schemes, communication scales linearly with the database size due to the growing query dimension, but is insensitive to block size since the query dominates the cost.

Figure 7(b) presents the client-server communication during eviction. Under the same conditions (2^{17} entries, 64-byte blocks), DuetORAM On requires about 0.76 MiB, significantly less than S^{3}ORAM's 18.6 MiB, because it transmits a vector rather than a matrix. In DuetORAM On, the client transmits shuffle correlations $(\boldsymbol{\Delta}, \boldsymbol{a}, \boldsymbol{b})$ of length $Z(H+2)$ with block size B, rendering its communication overhead explicitly *block-size dependent*. In contrast, DuetORAM Off utilizes offline preprocessing to decouple online communication from the block size. During the online phase, the client only needs to transmit the cyclic shift vectors $(\boldsymbol{r}_0, \boldsymbol{r}_1)$, whose elements are completely *block-size independent*, thereby drastically minimizing the online-only client-server communication. To quantify this



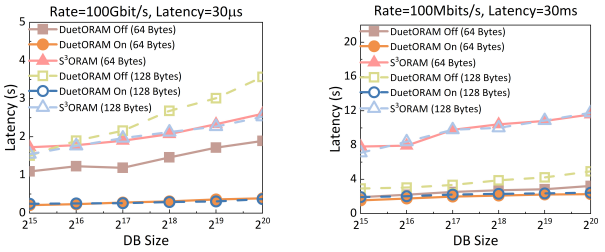
(a) LAN retrieval latency (b) WAN retrieval latency (c) LAN retrieval latency (d) WAN retrieval latency

Figure 6: Comparison of retrieval performance between DuetORAM and its counterparts.



(a) Client-server cost per retrieval (b) Client-server cost per eviction (c) Server-server cost per eviction

Figure 7: Communication overhead comparison between DuetORAM and its counterpart.



(a) LAN setting (b) WAN setting

Figure 8: Eviction performance comparison.

scaling advantage for large blocks, when $N = 2^{17}$, $Z = 333$ and $B = 1$ KiB, the shuffle correlations $(\boldsymbol{\Delta}, \boldsymbol{a}, \boldsymbol{b})$ required by DuetORAM On scale up to 11.7 MiB, while the block-size independent cyclic shifts $(\boldsymbol{r}_0, \boldsymbol{r}_1)$ in DuetORAM Off consume merely 62.4 KiB. The trade-off for this online bandwidth reduction is the proactive transmission of a punctured seed matrix during the offline phase. However, because its elements are composed of compact cryptographic seeds rather than full data payloads, they remain strictly *block-size independent*, ensuring scalability for the *total communication* profile.

Figure 7(c) shows the inter-server communication during eviction. DuetORAM incurs roughly half the overhead of S^{3}ORAM, primarily because S^{3}ORAM requires all-to-all server interaction for degree reduction, leading to substantially higher communication. Unlike retrieval, eviction communication in both schemes scales with the block size, as this phase involves transferring and processing full data blocks.

6 Conclusion

We design and implement DuetORAM, a concretely efficient 2-server DORAM that achieves $O(\log N)$ blocks of inter-server communication in constant rounds, low client-server communication, and no linear server-side work, using only lightweight symmetric primitives. This is enabled by a new replicated-to-shared block encoding and an offline-online decomposed secret-shared shuffle protocol. We evaluate DuetORAM against prior multi-server ORAM schemes under diverse network conditions. Our results show that, in LAN settings, DuetORAM reduces retrieval latency by up to two orders of magnitude compared to prior 2-server DORAM constructions and even outperforms 3-server designs. Moreover, DuetORAM achieves consistently lower eviction latency than 3-server ORAM in both LAN and WAN environments.

7 Acknowledgment

We thank the anonymous reviewers and shepherd for their insightful suggestions and comments. This work was supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office. It was also supported in part by the National Natural Science Foundation of China under Grant No. 62302118 and No. 62402155; and in part by the Zhejiang Provincial Natural Science Foundation of China under Grant No. LQ24F020013. Any opinions, findings, and conclusions

or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

Ethical Considerations

This work focuses on the design and system performance of cryptographic protocols. All experiments were conducted using synthetically generated datasets. Our study does not involve human subjects, personally identifiable information, or interactions with real-world production systems.

Stakeholders. The primary stakeholders include data owners, cloud service providers, and researchers in data privacy. Data owners benefit from stronger protection of both data content and access patterns when outsourcing storage and computation. Cloud service providers may enhance user trust and better align their services with privacy regulations by adopting privacy-preserving technologies. Researchers may build upon our techniques to advance their work or adopt our approach to address privacy challenges in related domains.

Ethical Principles. We consider our work under the four principles of the Menlo Report. Under *Respect for Persons*, we respect individuals by preventing inference of their data access behavior, thereby preserving autonomy and privacy. Under *Beneficence*, our work advances the privacy-preserving technology by improving the efficiency and practicality of distributed ORAM, contributing to more secure data outsourcing systems. Under *Justice*, our design lowers the cost of privacy protection, making such techniques more accessible, while introducing trust assumptions that must be carefully managed. Under *Respect for Law and Public Interest*, all experiments are conducted on synthetic data.

Potential Harms and Mitigations. The security of DuetORAM relies on a standard non-collusion assumption between the two servers. While this assumption is widely adopted in distributed ORAM literature, its violation, such as through collusion of the servers or compromise, would break privacy guarantees and expose both data contents and access patterns. To mitigate this risk, we explicitly state this assumption and emphasize its importance for security. We recommend deployment in settings where non-collusion is realistic, such as across independent cloud providers with economic or legal separation.

Decision to Proceed and Publish. We proceed with publication for several reasons. 1) *Technical novelty*: We introduce a replicated-to-shared block encoding that enables servers to locally convert identical ciphertexts into secret shares without interaction, which may inspire further research in efficient secure computation. 2) *Positive impact*: Our work improves the efficiency of two-server DORAM, reducing communication overhead and latency, thereby making privacy-preserving technologies more practical and offering valuable insights and techniques for future research. 3) *Adherence to ethical principles*: Our research process followed ethical principles, ensuring no harm to any party during the discovery or evaluation phases. Overall, we believe that the benefits of advancing efficient privacy-preserving technologies outweigh the potential risks. The publication of this work contributes to a more secure and privacy-preserving digital ecosystem.

Open Science

To support open science and reproducible research, we have made our source code available at https://github.com/lifeng10/DuetORAM and https://doi.org/10.5281/zenodo.20568623.

References

[1] Technology deep dive: Building a faster oram layer for enclaves. https://signal.org/blog/building-faster-oram/. Accessed: 2026-01-10.

[2] Utahfs: Encrypted file storage. https://blog.cloudflare.com/utahfs/. Accessed: 2026-01-10.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: optimal oblivious ram. In *EUROCRYPT*, pages 403–432, 2020.

[4] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, pages 535–548, 2013.

[5] Léonard Assouline and Brice Minaud. Weighted oblivious ram, with applications to searchable symmetric encryption. In *EUROCRYPT*, pages 426–455, 2023.

[6] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, pages 668–679, 2015.

[7] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. Graphos: Towards oblivious graph processing. *Cryptology ePrint Archive*, 2024.

[8] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *ASIACRYPT*, pages 342–372, 2020.

[9] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A Yavuz. Titanium: A metadata-hiding file-sharing system with malicious security. In *NDSS*, 2022.

[10] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. In *NDSS*, 2020.

[11] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[12] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, pages 79–88, 2006.

[13] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, pages 2433–2450, 2020.

[14] Srinivas Devadas, Marten Van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *TCC*, pages 145–174, 2015.

[15] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*, pages 523–535, 2017.

[16] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. Gigadoram: Breaking the billion address barrier. In *USENIX Security*, pages 3871–3888, 2023.

[17] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: round-optimal oblivious ram with applications to searchable encryption. *Cryptology ePrint Archive*, 2015.

[18] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *PETs*, pages 1–18, 2013.

[19] Tim Geoghegan, Christopher Patton, Eric Rescorla, and Christopher A Wood. Distributed aggregation protocol for privacy preserving measurement. *IETF*, 2023.

[20] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[21] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server oram. In *ASIACRYPT*, pages 141–157, 2018.

[22] Ariel Hamlin and Mayank Varia. Two-server distributed oram with sublinear computation and constant rounds. In *PKC*, pages 499–527, 2021.

[23] Thang Hoang, Jorge Guajardo, and Attila A Yavuz. Macao: A maliciously-secure and client-efficient active oram framework. *Cryptology ePrint Archive*, 2020.

[24] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S^3ORAM: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In *CCS*, pages 491–505, 2017.

[25] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-supported oram in effect: Practical oblivious search and update on very large dataset. *PETs*, 2019(1), 2019.

[26] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[27] Tung Le, Rouzbeh Behnia, Jorge Guajardo, and Thang Hoang. MUSES: Efficient Multi-User searchable encrypted database. In *USENIX Security*, pages 2581–2598, 2024.

[28] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S&P*, pages 359–376, 2015.

[29] Zheli Liu, Yanyu Huang, Xiangfu Song, Bo Li, Jin Li, Yali Yuan, and Changyu Dong. Eurus: Towards an efficient searchable symmetric encryption with size pattern protection. *TDSC*, 19(3):2023–2037, 2020.

[30] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*, 2014.

[31] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *S&P*, pages 279–296. IEEE, 2018.

[32] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security*, pages 415–430, 2015.

[33] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: Closing the gap between small and large client storage oblivious ram. *IACR Cryptol. ePrint Arch.*, 2014:997, 2014.

[34] Elaine Shi, T H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[35] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *CCS*, pages 247–258, 2013.

[36] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[37] 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 *CCS*, pages 299–310, 2013.

[38] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A Bandwidth-Efficient distributed ORAM for 2- and 3-party computation. In *USENIX Security*, pages 3907–3924, 2023.

[39] Wei Wang, Xianglong Zhang, Peng Xu, Rongmao Chen, and Laurence Tianruo Yang. Bandwidth-efficient two-server orams with o (1) client storage. *arXiv preprint arXiv:2503.21126*, 2025.

[40] Zhiqiang Wu and Rui Li. Obi: a multi-path oblivious ram for forward-and-backward-secure searchable encryption. In *NDSS*, 2023.

[41] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: the power of File-Injection attacks on searchable encryption. In *USENIX Security*, pages 707–720, 2016.

A Comparison Table

Table 2 summarizes the theoretical cost comparison between DuetORAM and prior work.

B Correctness and Security Definition for PIR

Correctness. Correctness requires that for any database DB with $n = |\mathsf{DB}|$, and query $i \in [n]$, the PIR reconstruction can recover the correct data item $\mathsf{DB}[i]$ with probability 1.

$$\Pr\left[\begin{array}{l}(\boldsymbol{q}_0,\boldsymbol{q}_1)\leftarrow \mathsf{Create}(i),\\ a_0\leftarrow \mathsf{Retrieve}(0,\boldsymbol{q}_0,\mathsf{DB}),\\ a_1\leftarrow \mathsf{Retrieve}(1,\boldsymbol{q}_1,\mathsf{DB}),\\ \qquad \mathsf{Reconstruct}(a_0,a_1)=\mathsf{DB}[i]\end{array}\right]=1$$

Security. We define the security of two-server PIR in the Real/Ideal simulation paradigm with adaptive PIR queries, adapting the standard PIR security definition [11]. Security requires that any PPT adversary $\mathcal{A}$ who corrupts only one server S_t for $t \in \{0,1\}$ learns no information, which is captured by the following Real/Ideal security definition.

- $\mathsf{Real}_t^{\mathcal{A}}(1^\lambda, n)$: On receiving an index $i \in [n]$ from $\mathcal{A}$, the client executes $(\boldsymbol{q}_0,\boldsymbol{q}_1) \leftarrow \mathsf{Create}(i)$ and sends $\boldsymbol{q}_0$ and $\boldsymbol{q}_1$ to server 0 and server 1, respectively. Each server S_t executes $a_t \leftarrow \mathsf{Retrieve}(t,\boldsymbol{q}_t,\mathsf{DB})$. The client obtains the query result $\mathsf{DB}[i] = \mathsf{Reconstruct}(a_0,a_1)$. The above game can be run polynomially many times. At the end of the game, $\mathcal{A}$ outputs a bit $guess \in \{0,1\}$.
- $\mathsf{Ideal}_t^{\mathcal{A},\mathcal{S}}(1^\lambda, n)$: On receiving an index $i \in [n]$ from $\mathcal{A}$, the simulator $\mathcal{S}$ works as $\boldsymbol{q}_t \leftarrow \mathcal{S}(1^\lambda, t, n)$ and sends $\boldsymbol{q}_t$ to server S_t (thus $\mathcal{A}$). Each server S_t executes $a_t \leftarrow \mathsf{Retrieve}(t,\boldsymbol{q}_t,\mathsf{DB})$. The client obtains the query result $\mathsf{Reconstruct}(a_0,a_1)$. The above game can be run polynomially many times. At the end of the game, $\mathcal{A}$ outputs a bit $guess \in \{0,1\}$.

In the above Real/Ideal games, we allow the adversary to choose queries adaptively. We say Π_{PIR} is *adaptively secure* if for any PPT adversary $\mathcal{A}$ who corrupts one single server S_t, there is a PPT simulator $\mathcal{S}$ such that Real and Ideal are computationally indistinguishable:

$$\left|\Pr\left[\mathsf{Real}_t^{\mathcal{A}}(1^\lambda,n)=1\right]-\Pr\left[\mathsf{Ideal}_t^{\mathcal{A},\mathcal{S}}(1^\lambda,n)=1\right]\right|\le \mathsf{negl}(\lambda).$$

C Correctness and Security Definition of DuetORAM

In this paper, we consider a system model consisting of a trusted client and two semi-honest, non-colluding servers. Under this model, DuetORAM aims to achieve oblivious access, meaning that any individual server, based solely on its local view, cannot infer any information about the client's access pattern, including the accessed block identifiers, access locations, or the operation types (i.e., read or write).

To formally define the correctness and security of DuetORAM, we introduce the following notation. Let

$$\mathbf{x} = \{(\mathsf{op}_i, \mathsf{id}_i, \mathsf{data}_i)\}_{i=1}^{M}$$

denote an access sequence of length M, where $\mathsf{op}_i \in \{\mathsf{read}, \mathsf{write}\}$ specifies the type of the i-th operation, id_i denotes the identifier of the accessed data block, and data_i represents the data to be written when $\mathsf{op}_i = \mathsf{write}$.

C.1 Correctness Definition of DuetORAM

Let $ORAM_t(\mathbf{x})$ denote the randomized access transcript between the client and server S_t induced by executing the access sequence $\mathbf{x}$. Based on these definitions, we next formalize the correctness and security guarantees of DuetORAM.

Definition 2 (Correctness of DuetORAM)**.** DuetORAM *is correct if for any security parameter $\lambda \in \mathbb{N}$, any system parameters that are correctly instantiated according to Lemma 1, and any access sequence $\mathbf{x}$, we have*

$$\Pr[\mathsf{DuetORAM}(\mathbf{x}) \neq \mathsf{RAM}(\mathbf{x})] \le \mathsf{negl}(\lambda),$$

where the probability is taken over the randomness of the protocol, $\mathsf{DuetORAM}(\mathbf{x})$ *denotes the output recovered by the client after executing the* DuetORAM *protocol on input* $\mathbf{x}$*,*

Table 2: Theoretical Cost Comparison of ORAM Schemes (Secure Computation ORAMs Interpreted as Single-Client Designs)

Schemes	Servers	Retrieval				Eviction		
		C-S Comm.	S-S Comm.	Client Comp.	Server Comp.	C-S Comm.	S-S Comm.	S-S Round
Path ORAM [37]	1	$O(B\cdot Z\cdot H)$	–	$O(Z\cdot H)$	$O(1)$	–	–	–
FLORAM [15]	2	$O(\lambda\cdot H)$	–	$O(H)$	$O(N+\sqrt{N})$	$O(\lambda)$	$O(N\cdot B)$	$O(1)$
DUORAM [38]	2 or 3	$O(\lambda\cdot H)$+$O(B)$	$O(B)$+$O(B)$	$O(H)$+$O(N)$	$O(N)$+$O(N)$	–	–	–
S^3ORAM [24]	≥ 3	$O(\mu\cdot Z\cdot H+B)$	–	$O(Z\cdot H)$	$O(Z\cdot H)$	$O(\mu\cdot Z^2\cdot H)$	$O(Z\cdot B\cdot H)$	$O(H)$
Ours	2	$O(Z\cdot H+B)$	–	$O(Z\cdot H)$	$O(Z\cdot H)$	$O(\lambda\cdot Z^2\cdot H^2)$+ $O(Z\cdot H\cdot\log(Z\cdot H))$	$O(Z\cdot B\cdot H)$	$O(1)$

Note: To present a theoretically fair comparison, we treat ORAM schemes in the context of secure computation (e.g., FLORAM and DUORAM) as if the functionality that collectively emulates the ORAM client is executed by a single client. In other words, computations originally performed jointly by multiple servers via secure computation protocols are conceptually attributed to the client. FLORAM does not include an eviction operation, but it requires a periodic procedure (called refresh) that serves a similar purpose. Therefore, we group them together in the table. μ denotes the size of a chunk. The gray color represents the offline cost.

$\mathsf{RAM}(\mathbf{x})$ *denotes the output of executing the same access sequence in an ideal RAM, and* $\mathsf{negl}(\lambda)$ *is a negligible function in the security parameter* λ.

C.2 Security Definition of DuetORAM

Threat Model. We consider a system model comprising a trusted client and two non-colluding, semi-honest servers, denoted as S_0 and S_1. Our threat model follows the standard framework used in distributed secure computation and oblivious storage. A PPT adversary $\mathcal{A}$ can statically corrupt at most one of the two servers S_t ($t\in\{0,1\}$). Under the semi-honest assumption, the corrupted server faithfully follows the protocol specification but attempts to infer sensitive information by inspecting its internal view. The view of a corrupted server S_t includes its private PRF key, the local encrypted storage, and the transcripts of all incoming messages from the client or the peer server during protocol execution.

Our goal is to ensure oblivious access, meaning that the adversary, based solely on its local view, cannot infer any information about the client's access patterns or the underlying data contents. Specifically, the access pattern must hide: (1) the identities of the accessed blocks, (2) the physical access locations within the tree structure, and (3) the specific operation types (*i.e.*, read or write). Security is modeled via a simulation-based definition, where the Real and Ideal games are defined as follows:

- $\mathsf{Real}_t^{\mathcal{A}}(1^\lambda,N)$: On receiving a database DB from $\mathcal{A}$, the game executes $\mathsf{Setup}(1^\lambda,\mathsf{DB})$ and sends EDB_t to $\mathcal{A}$. After that, $\mathcal{A}$ can adaptively make polynomially many access queries. Whenever $\mathcal{A}$ issues an access query $(\mathsf{op},\mathsf{id},\mathsf{data})$, the game executes $\mathsf{Access}(\sigma,\mathsf{op},\mathsf{id},\mathsf{data})$. We use view to denote all transcripts that correspond to the server S_t, including the messages from client to server S_t and the messages from server S_{1-t} to server S_t. The adversary obtains all transcripts that correspond to the server S_t. At the end of the game, $\mathcal{A}$ outputs a bit $guess\in\{0,1\}$.
- $\mathsf{Ideal}_t^{\mathcal{A},\mathcal{S}}(1^\lambda,N)$: On receiving a database DB from $\mathcal{A}$, the game call simulator $\mathcal{S}$ to obtain $(\sigma,\mathsf{EDB}_t)\leftarrow\mathcal{S}(1^\lambda,|\mathsf{DB}|)$ and sends EDB_t to $\mathcal{A}$. After that, $\mathcal{A}$ can adaptively make polynomially many access queries. Whenever $\mathcal{A}$ issues an access query $(\mathsf{op},\mathsf{id},\mathsf{data})$, the game call $(\sigma,\mathsf{view})\leftarrow\mathcal{S}(\sigma,|\mathsf{DB}|)$, where view contains all transcripts that correspond to the server S_t, including the messages from client to server S_t and the messages from server S_{1-t} to server S_t. At the end of the game, $\mathcal{A}$ outputs a bit $guess\in\{0,1\}$.

Definition 3 (Two-server DORAM Security)**.** *We say a two-server DORAM scheme* Π_{DORAM} *is adaptively secure if for any PPT adversary* $\mathcal{A}$ *who corrupts one single server* S_t*, there is a PPT simulator* $\mathcal{S}$ *such that* Real *and* Ideal *are computationally indistinguishable:*

$$\left|\Pr\left[\mathsf{Real}_t^{\mathcal{A}}(1^\lambda,N)=1\right]-\Pr\left[\mathsf{Ideal}_t^{\mathcal{A},\mathcal{S}}(1^\lambda,N)=1\right]\right|\leq\mathsf{negl}(\lambda).$$

C.3 Definitions for $\mathcal{F}_{\mathsf{perm}}$

To support the modular analysis of our design, we formalize the correctness and security requirements for a concrete client-aided 2-server permutation protocol Π_{perm}. Concretely, let $\Pi_{\mathsf{perm}}\in\{\Pi_{\mathsf{perm\text{-}on}},\Pi_{\mathsf{perm\text{-}off}}\}$, where $\Pi_{\mathsf{perm\text{-}on}}$ denotes our pure-online instantiation (Section 3.5.1) and $\Pi_{\mathsf{perm\text{-}off}}$ denotes our offline-online instantiation (Section 3.5.2), both aiming to realize the ideal 3-party standalone functionality $\mathcal{F}_{\mathsf{perm}}$. In this framework, the trusted client supplying the target permutation π is considered uncorrupted, and $\mathcal{A}$ be a PPT adversary can statically corrupt at most one server S_t ($t\in\{0,1\}$).

Definition 4 (Correctness of $\mathcal{F}_{\mathsf{perm}}$ Instantiation)**.** *A client-aided 2-server permutation protocol* Π_{perm} *is correct if for any target permutation* $\pi\in\mathbf{S}_n$ *chosen by the client, and any input vector* $\boldsymbol{v}$ *initially XOR-shared between the two servers as* $\boldsymbol{v}=[\![\boldsymbol{v}]\!]_0\oplus[\![\boldsymbol{v}]\!]_1$*, the respective outputs* $[\![\tilde{\boldsymbol{v}}]\!]_0$ *and* $[\![\tilde{\boldsymbol{v}}]\!]_1$ *obtained*

by the servers at the end of the protocol execution strictly satisfy:

$$[\![\tilde{\boldsymbol{v}}]\!]_0 \oplus [\![\tilde{\boldsymbol{v}}]\!]_1 = \pi(\boldsymbol{v})$$

with probability 1.

We now define the security of functionality $\mathcal{F}_{\mathsf{perm}}$. Let S_t be the server that is corrupted, and let $\mathcal{S}$ denote a simulator. We define the following games:

- $\mathsf{Real}_t^{\mathcal{A}}(1^\lambda, \pi, [\![\boldsymbol{v}]\!]_0, [\![\boldsymbol{v}]\!]_1)$: Outputs a bit $guess \in \{0,1\}$ returned by $\mathcal{A}$ after observing the real execution view, which comprises all transcripts corresponding to server S_t, including the messages from the client to server S_t and the messages from server S_{1-t} to server S_t.
- $\mathsf{Ideal}_t^{\mathcal{A},\mathcal{S}}(1^\lambda, \pi, [\![\boldsymbol{v}]\!]_0, [\![\boldsymbol{v}]\!]_1)$: Outputs a bit $guess \in \{0,1\}$ returned by $\mathcal{A}$ after observing the simulated view generated by $\mathcal{S}_t(1^\lambda, [\![\boldsymbol{v}]\!]_t, [\![\tilde{\boldsymbol{v}}]\!]_t)$, where $[\![\tilde{\boldsymbol{v}}]\!]_t$ is the ideal output share received from $\mathcal{F}_{\mathsf{perm}}$.

Definition 5 (Security of $\mathcal{F}_{\mathsf{perm}}$ Instantiation)**.** *We say* Π_{perm} *securely realizes* $\mathcal{F}_{\mathsf{perm}}$ *if for any PPT adversary* $\mathcal{A}$ *who corrupts one single server* S_t*, there is a PPT simulator* $\mathcal{S}$ *such that* Real *and* Ideal *are computationally indistinguishable:*

$$\left|\Pr\left[\mathsf{Real}_t^{\mathcal{A}}(1^\lambda, \pi, [\![\boldsymbol{v}]\!]_0, [\![\boldsymbol{v}]\!]_1) = 1\right] - \Pr\left[\mathsf{Ideal}_t^{\mathcal{A},\mathcal{S}}(1^\lambda, \pi, [\![\boldsymbol{v}]\!]_0, [\![\boldsymbol{v}]\!]_1) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

D Correctness and Security Proof

D.1 Correctness Proof

Lemma 2 (Equivalence of Path Eviction Strategies)**.** *Suppose the system parameters satisfy Lemma 1 and eviction paths are chosen in reverse lexicographical order. Then performing a single greedy eviction over all blocks on the eviction path is logically equivalent to executing triplet eviction level by level.*

Proof. By Lemma 1, the probability that a bucket overflows during any bucket triplet eviction step is negligible. Hence, throughout the eviction process, we assume that all buckets can accommodate the blocks that are assigned to them.

We compare the following two eviction strategies applied to the same path:

1. **Level-by-level triplet eviction**: bucket triplet eviction is executed sequentially from the root to the leaf along the eviction path.
2. **Single-shot greedy eviction**: all blocks on the eviction path are considered together and are greedily pushed downward to the deepest buckets in which they can legally reside.

For any block b, let d_b denote the depth of the deepest bucket on the eviction path that is consistent with the leaf assigned to b. Note that d_b depends only on the block's assigned leaf and the eviction path, and is independent of the eviction strategy.

We prove that both strategies place every block into the same final bucket by induction on the depth $l = 0, 1, \cdots, L$, where L is the leaf depth. We maintain the invariant that after processing depth l, in both strategies each block has been placed in the deepest bucket it can reach, and that bucket is at depth at most l.

Step 1: Base case ($l = 0$). Initially, all blocks reside in the root bucket in both strategies. The invariant trivially holds.

Step 2: Inductive step. Assume the invariant holds after processing depth $l-1$. Consider any block b.

- If $d_b < l$, then b cannot be pushed further down the eviction path. In level-by-level triplet eviction, it is routed into the sibling bucket at depth d_b. In the greedy strategy, the block likewise cannot be placed into any bucket deeper than d_b, and therefore remains at the same depth.
- If $d_b \geq l$, then b can legally reside in the bucket at depth l on the eviction path. In level-by-level triplet eviction, when processing the triplet at depth l, the block is moved into the destination bucket at that level. In the greedy strategy, since Lemma 1 guarantees that bucket capacities suffice for all blocks that are legally assigned to that level except with negligible probability, the greedy procedure also places b into that bucket.

Moreover, because bucket capacities are sufficient for all legally placable blocks except with negligible probability, the greedy strategy never blocks a placement that would succeed under the level-by-level eviction. Thus, after processing depth l, the placement of every block is identical in both strategies, and the invariant continues to hold.

When $l = L$, every block has been placed in the deepest bucket it can legally occupy. Therefore, both eviction strategies result in the same final block distribution. Hence, performing a single greedy eviction over the entire path is logically equivalent to executing bucket triplet eviction level by level. □

Corollary 1 (Correctness of Path Eviction)**.** *By Lemma 2 and the correctness of level-by-level triplet eviction proven in prior constructions [14, 24], the single-shot eviction procedure used in* DuetORAM *satisfies the standard ORAM eviction invariant: after eviction, each block resides in the deepest bucket along its assigned path that is consistent with capacity constraints, except with negligible probability.*

We present separate correctness lemmas for our two distinct client-aided protocols $\Pi_{\mathsf{perm\text{-}on}}$ and $\Pi_{\mathsf{perm\text{-}off}}$.

Lemma 3 (Correctness of $\Pi_{\mathsf{perm\text{-}on}}$)**.** *The pure-online client-aided permutation protocol is correct under Definition 4.*

Proof. By the protocol specification of $\Pi_{\text{perm-on}}$, the client ensures that $\pi = \pi_0 \circ \pi_1$ and computes $\boldsymbol{\Delta}_0 = \pi_0(\boldsymbol{a}_1) \oplus \boldsymbol{b}_1$ and $\boldsymbol{\Delta}_1 = \pi_1(\boldsymbol{a}_0) \oplus \boldsymbol{b}_0$. The client then sends $(\pi_0, \boldsymbol{\Delta}_0, \boldsymbol{a}_0, \boldsymbol{b}_0)$ to S_0 and $(\pi_1, \boldsymbol{\Delta}_1, \boldsymbol{a}_1, \boldsymbol{b}_1)$ to S_1. For the shares $[\![\boldsymbol{v}]\!]_0$ and $[\![\boldsymbol{v}]\!]_1$ held by S_0 and S_1:

1. Server S_0 computes $\boldsymbol{x}_0^{(2)} = \pi_0(\boldsymbol{x}_0^{(1)}) \oplus \boldsymbol{\Delta}_0 \oplus \pi_0([\![\boldsymbol{v}]\!]_0)$. Substituting $\boldsymbol{\Delta}_0 = \pi_0(\boldsymbol{a}_1) \oplus \boldsymbol{b}_1$ and $\boldsymbol{x}_0^{(1)} = \boldsymbol{a}_1 \oplus [\![\boldsymbol{v}]\!]_1$ (received from S_1):
$$\begin{aligned}\boldsymbol{x}_0^{(2)} &= \pi_0(\boldsymbol{a}_1 \oplus [\![\boldsymbol{v}]\!]_1) \oplus (\pi_0(\boldsymbol{a}_1) \oplus \boldsymbol{b}_1) \oplus \pi_0([\![\boldsymbol{v}]\!]_0) \\ &= \pi_0([\![\boldsymbol{v}]\!]_0 \oplus [\![\boldsymbol{v}]\!]_1) \oplus \boldsymbol{b}_1 = \pi_0(\boldsymbol{v}) \oplus \boldsymbol{b}_1.\end{aligned}$$
2. Server S_1 computes $\boldsymbol{x}_1^{(2)} = \pi_1(\boldsymbol{x}_1^{(1)}) \oplus \boldsymbol{\Delta}_1 \oplus \pi_1(\boldsymbol{b}_1) \oplus \boldsymbol{w}$. Substituting $\boldsymbol{\Delta}_1 = \pi_1(\boldsymbol{a}_0) \oplus \boldsymbol{b}_0$ and $\boldsymbol{x}_1^{(1)} := \boldsymbol{a}_0 \oplus \boldsymbol{x}_0^{(2)}$ (received from S_0):
$$\begin{aligned}\boldsymbol{x}_1^{(2)} &= \pi_1(\boldsymbol{a}_0 \oplus \boldsymbol{x}_0^{(2)}) \oplus (\pi_1(\boldsymbol{a}_0) \oplus \boldsymbol{b}_0) \oplus \pi_1(\boldsymbol{b}_1) \oplus \boldsymbol{w} \\ &= \pi_1(\boldsymbol{a}_0 \oplus \pi_0(\boldsymbol{v}) \oplus \boldsymbol{b}_1) \oplus \pi_1(\boldsymbol{a}_0) \oplus \boldsymbol{b}_0 \oplus \pi_1(\boldsymbol{b}_1) \oplus \boldsymbol{w} \\ &= \pi_1(\pi_0(\boldsymbol{v})) \oplus \boldsymbol{b}_0 \oplus \boldsymbol{w} = \pi(\boldsymbol{v}) \oplus \boldsymbol{b}_0 \oplus \boldsymbol{w}.\end{aligned}$$

Combining S_0's designated output share ($[\![\tilde{\boldsymbol{v}}]\!]_0 = \boldsymbol{b}_0 \oplus \boldsymbol{w}$) and S_1's output share ($[\![\tilde{\boldsymbol{v}}]\!]_1 = \boldsymbol{x}_1^{(2)}$) yields: $[\![\tilde{\boldsymbol{v}}]\!]_0 \oplus [\![\tilde{\boldsymbol{v}}]\!]_1 = \boldsymbol{b}_0 \oplus \boldsymbol{w} \oplus \pi(\boldsymbol{v}) \oplus \boldsymbol{b}_0 \oplus \boldsymbol{w} = \pi(\boldsymbol{v})$. This matches the exact correctness criteria of Definition 4 with probability 1. □

Lemma 4 (Correctness of $\Pi_{\text{perm-off}}$)**.** *The optimized offline-online permutation protocol is correct under Definition 4.*

Proof. In the offline phase of $\Pi_{\text{perm-off}}$, row i of the pre-distributed matrix $\boldsymbol{A}$ is punctured at position $\boldsymbol{u}_0[i]$. In the online phase, the client provides the cyclic shift vector $\boldsymbol{r}_0[i] = (\pi_0[i] - \boldsymbol{u}_0[i]) \bmod n$. When S_0 applies a row-wise right cyclic shift of $\boldsymbol{r}_0[i]$ steps, the updated puncture index $\boldsymbol{u}'_0[i]$ becomes:
$$\boldsymbol{u}'_0[i] = (\boldsymbol{u}_0[i] + \boldsymbol{r}_0[i]) \bmod n.$$
Substituting $\boldsymbol{r}_0[i]$:
$$\boldsymbol{u}'_0[i] = (\boldsymbol{u}_0[i] + (\pi_0[i] - \boldsymbol{u}_0[i])) \bmod n = \pi_0[i].$$

Symmetrically, the same mapping shifts the puncture of matrix $\boldsymbol{B}'$ to exactly $\pi_1[i]$ for S_1. Consequently, after the local row-wise cyclic shifts, the internal state of the servers' expanded matrices aligns perfectly with the puncture requirements of the standard CGP Share Translation protocol evaluated at sub-permutations π_0 and π_1. The remaining interactive execution reduces identically to the algebraic equations in Lemma 3, thereby inheriting the same correctness guarantee for $\Pi_{\text{perm-off}}$. □

Theorem 2 (Correctness of DuetORAM)**.** *For any security parameter $\lambda \in \mathbb{N}$, any system parameters that are correctly instantiated according to Lemma 1, and any access sequence* $\mathbf{x}$*, the output recovered by the client after executing* $\mathsf{DuetORAM}(\mathbf{x})$ *is identical to the output of executing* $\mathbf{x}$ *in an ideal RAM, except with negligible probability in* λ*.*

Proof. We now formally prove that DuetORAM satisfies the correctness guarantee in Definition 2. Correctness follows from (i) correct retrieval of the requested block and (ii) correct eviction that preserves the ORAM structural invariant.

Retrieval Correctness. Consider an access request $\mathbf{x}_i = (\mathsf{op}_i, \mathsf{id}_i, \mathsf{data}_i)$. The client first looks up the position map to obtain the leaf label and path index $(\mathsf{pID}_i, \mathsf{pIdx}_i) \leftarrow \mathsf{pm}[\mathsf{id}_i]$. By the DuetORAM invariant maintained during eviction, the ciphertext corresponding to block id_i is guaranteed to reside in pIdx_i along path pID_i.

The retrieval phase is implemented as a two-server PIR over the replicated ciphertext blocks stored on path pID_i, where the client queries for index pIdx_i. By the correctness of the PIR protocol, the client obtains the ciphertext
$$c = b \oplus F(k_0, \eta_0) \oplus F(k_1, \eta_1),$$
where b is the underlying plaintext block. The client can locally recompute the PRF masks $F(k_0, \eta_0)$ and $F(k_1, \eta_1)$, and remove them to recover
$$b = c \oplus F(k_0, \eta_0) \oplus F(k_1, \eta_1).$$
Thus, the client obtains exactly the plaintext block stored at the path index pIdx_i, which corresponds to the block id_i. Therefore, the retrieval procedure is correct.

For write operations, the client updates the block value to data_i (or keeps b for reads) and re-encrypts it before writing it back to the root bucket, after which the standard eviction procedure restores the ORAM invariant.

Eviction Correctness. Corollary 1 implies that the permutation π induced by the single-shot greedy eviction strategy correctly realizes a semantically valid ORAM path eviction. By the hybrid composition theorem, we analyze eviction in the $\mathcal{F}_{\text{perm}}$-hybrid model. Assuming $\mathcal{F}_{\text{perm}}$ is an ideal functionality, it takes the target eviction permutation π from the client and outputs valid permuted plaintext shares satisfying $[\![\tilde{\boldsymbol{v}}]\!]_0 \oplus [\![\tilde{\boldsymbol{v}}]\!]_1 = \pi(\boldsymbol{v})$. During the write-back phase (Protocol 4, Steps $5-7$), each server masks its share with a fresh nonce η_i expanded via a PRG. The servers exchange these blinded shares and convert permuted blocks into identical double-masked format for storage.

Finally, by substituting either Lemma 3 ($\Pi_{\text{perm-on}}$) or Lemma 4 ($\Pi_{\text{perm-off}}$) into the $\mathcal{F}_{\text{perm}}$-hybrid model, the real-world eviction successfully restores the structural ORAM invariant, ensuring each block resides in its deepest valid bucket except with a negligible bucket overflow probability.

In summary, since every access retrieves the correct block and eviction maintains a valid ORAM structure for future accesses, the sequence of operations in $\mathsf{DuetORAM}(\mathbf{x})$ produces exactly the same outputs as an ideal RAM execution, except with negligible probability arising from the negligible bucket overflow probability in Lemma 1.

□

D.2 Security Proof

Lemma 5. *The pure-online client-aided permutation protocol* $\Pi_{\mathsf{perm\text{-}on}}$ *securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{perm}}$ *under Definition 5 if hash function H is modeled as a random oracle.*

Proof. For any PPT adversary $\mathcal{A}$ who statically corrupts S_t, we construct distinct PPT simulators for the two separate corruption cases, accounting for the algebraic asymmetry inherent in the protocol transcript.

Case 1: Corrupted Server S_0. In the case where S_0 is corrupted, the simulator works as follows:

$\mathcal{S}_{\mathsf{perm\text{-}on},0}(1^\lambda, [\![\boldsymbol{v}]\!]_0, [\![\tilde{\boldsymbol{v}}]\!]_0)$:

1: Sample a uniformly random sub-permutation $\pi_0 \xleftarrow{\$} \mathbf{S}_n$, and uniformly random vectors $\boldsymbol{a}_0, \boldsymbol{b}_0, \boldsymbol{\Delta}_0 \xleftarrow{\$} (\{0,1\}^B)^n$.
2: To simulate the first-round incoming message from S_1, sample a uniformly random vector $\boldsymbol{x}_0^{(1)} \xleftarrow{\$} (\{0,1\}^B)^n$.
3: Intercept k_w from $\mathcal{A}$ and program $H(k_w) := [\![\tilde{\boldsymbol{v}}]\!]_0 \oplus \boldsymbol{b}_0$.

We argue that the above simulation generates an execution view that is indistinguishable from the real-protocol execution. First, regarding the client-distributed parameters, $\pi_0, \boldsymbol{a}_0$, and $\boldsymbol{b}_0$ are sampled uniformly at random in the real protocol, perfectly matching the simulation. The real parameter $\boldsymbol{\Delta}_0 = \pi_0(\boldsymbol{a}_1) \oplus \boldsymbol{b}_1$ is uniformly distributed over $(\{0,1\}^B)^n$ from S_0's perspective because it is perfectly blinded by the independent random vector $\boldsymbol{b}_1$ which is strictly hidden from S_0.

Second, regarding the communication transcript, the real first-round message $\boldsymbol{x}_0^{(1)} = \boldsymbol{a}_1 \oplus [\![\boldsymbol{v}]\!]_1$ is uniformly distributed because the confidential random vector $\boldsymbol{a}_1$ acts as a perfect one-time pad over S_1's input, making it indistinguishable to the uniform random vector sampled by $\mathcal{S}_{\mathsf{perm\text{-}on},0}$. Third, the terminal output consistency is perfectly satisfied via Step 3. Since k_w is drawn from an exponentially large security space $\{0,1\}^\lambda$, the probability that $\mathcal{A}$ has queried the random oracle at k_w prior to this step is negligible. Thus, programming $H(k_w) := [\![\tilde{\boldsymbol{v}}]\!]_0 \oplus \boldsymbol{b}_0$ is perfectly indistinguishable from a true random response. As a result, S_0's terminal local output $\boldsymbol{b}_0 \oplus H(k_w)$ evaluates identically to the target ideal share $[\![\tilde{\boldsymbol{v}}]\!]_0$, completing the analysis for Case 1.

Case 2: Corrupted Server S_1. In the case where S_1 is corrupted, the simulator works as follows:

$\mathcal{S}_{\mathsf{perm\text{-}on},1}(1^\lambda, [\![\boldsymbol{v}]\!]_1, [\![\tilde{\boldsymbol{v}}]\!]_1)$:

1: Sample a uniformly random sub-permutation $\pi_1 \xleftarrow{\$} \mathbf{S}_n$, and uniformly random vectors $\boldsymbol{a}_1, \boldsymbol{b}_1 \xleftarrow{\$} (\{0,1\}^B)^n$.
2: To simulate the second-round incoming message from S_0, sample a uniformly random vector $\boldsymbol{x}_1^{(1)} \xleftarrow{\$} (\{0,1\}^B)^n$ and a uniform key $k_w \xleftarrow{\$} \{0,1\}^\lambda$, then query the random oracle to obtain $\boldsymbol{w} = H(k_w)$.
3: Backward-patch the remaining client-distributed parameter $\boldsymbol{\Delta}_1 := [\![\tilde{\boldsymbol{v}}]\!]_1 \oplus \pi_1(\boldsymbol{x}_1^{(1)}) \oplus \pi_1(\boldsymbol{b}_1) \oplus \boldsymbol{w}$ to enforce algebraic and output consistency.

By symmetry to Case 1, the uniform distributions of $(\pi_1, \boldsymbol{a}_1, \boldsymbol{b}_1)$ and the one-time pad property of the incoming transcript $\boldsymbol{x}_1^{(1)}$ (perfectly blinded by the hidden vector $\boldsymbol{a}_0$) follow identically. The key k_w is sampled uniformly at random, perfectly matching its distribution in the real execution. The only asymmetry lies in $\boldsymbol{\Delta}_1$, which is uniformly distributed in the real world due to the independent random share $\boldsymbol{b}_0$ strictly hidden from S_1. In the simulation, because the ideal output share $[\![\tilde{\boldsymbol{v}}]\!]_1$ is uniformly distributed, the backward-programmed parameter $\boldsymbol{\Delta}_1$ inherits perfect uniform randomness via the bijective nature of the XOR translation. Consequently, all simulated components maintain identical algebraic distributions to the real world, completing the proof. □

Lemma 6. *The optimized offline-online client-aided permutation protocol* $\Pi_{\mathsf{perm\text{-}off}}$ *securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{perm}}$ *under Definition 5 if G is a secure PRG and hash function H is modeled as a random oracle.*

Proof. For any PPT adversary $\mathcal{A}$ who statically corrupts S_t, we construct distinct PPT simulators for the two separate corruption cases.

Case 1: Corrupted Server S_0. In the case where S_0 is corrupted, the simulator works as follows:

$\mathcal{S}_{\mathsf{perm\text{-}off},0}(1^\lambda, [\![\boldsymbol{v}]\!]_0, [\![\tilde{\boldsymbol{v}}]\!]_0)$:

1: Sample a uniform PRG seed $k_4 \xleftarrow{\$} \{0,1\}^\lambda$ and a uniform puncturing vector $\boldsymbol{u}_0 \xleftarrow{\$} \{0,\ldots,n-1\}^n$.
2: Sample a uniform $n \times n$ matrix $\boldsymbol{A}$ such that $\boldsymbol{A}[i][j] \in \{0,1\}^\lambda$, and construct the punctured matrix $\boldsymbol{A}'$ by setting $\boldsymbol{A}'[i][\boldsymbol{u}_0[i]] \leftarrow 0$ for each row $i \in \{0,\ldots,n-1\}$ (where all other elements copy $\boldsymbol{A}$).
3: Generate an $n \times n$ matrix $\boldsymbol{B}$ by expanding the seed k_4 via the PRG G such that $\boldsymbol{B}[i][j] \in \{0,1\}^\lambda$.
4: Sample a uniformly random sub-permutation $\pi_0 \xleftarrow{\$} \mathbf{S}_n$ to derive the shift vector $\boldsymbol{r}_0[i] := (\pi_0[i] - \boldsymbol{u}_0[i]) \bmod n$. Symmetrically, sample a peer shift vector $\boldsymbol{r}_1 \xleftarrow{\$} \{0,\ldots,n-1\}^n$ uniformly at random.
5: Execute the row-wise cyclic shift operations by applying the shift vector $\boldsymbol{r}_0$ on $\boldsymbol{A}'$ and $\boldsymbol{r}_1$ on $\boldsymbol{B}$ to deterministically derive the online intermediate parameters, including $\boldsymbol{a}_0, \boldsymbol{b}_0$, and $\boldsymbol{\Delta}_0$.
6: Sample the online incoming transcript $\boldsymbol{x}_0^{(1)} \xleftarrow{\$} (\{0,1\}^B)^n$ uniformly at random.
7: Intercept k_w from $\mathcal{A}$ and program $H(k_w) := [\![\tilde{\boldsymbol{v}}]\!]_0 \oplus \boldsymbol{b}_0$.

We argue that the simulated view generated by $\mathcal{S}_{\mathsf{perm\text{-}off},0}$ is computationally indistinguishable from the real execution view. First, for the offline phase, the uniformly sampled seed k_4 is identically distributed to the real execution by construction. The punctured matrix $\boldsymbol{A}'$ in the simulation is generated by puncturing a truly random matrix at a uniformly chosen vector $\boldsymbol{u}_0$. Under the cryptographic security of the PRG G, the pseudo-random matrix expanded from the secret seed k_3

in the real protocol is computationally indistinguishable from this truly random matrix, ensuring the simulated offline view is indistinguishable from the real distribution.

Second, for the online packet, $\mathcal{S}_{\mathsf{perm\text{-}off},0}$ perfectly preserves the critical algebraic dependency $\boldsymbol{r}_0[i] = (\pi_0[i] - \boldsymbol{u}_0[i]) \bmod n$ derived from the punctured path $\boldsymbol{u}_0$. The peer's shift $\boldsymbol{r}_1$ is distributed uniformly at random in the real world because the confidential random path $\boldsymbol{u}_1$ acts as a perfect modular one-time pad over π_1, which matches the simulation.

Third, based on the aforementioned indistinguishable parameters, the local shuffle correlations $\boldsymbol{a}_0, \boldsymbol{b}_0$, and $\boldsymbol{\Delta}_0$ derived via online cyclic shifts perfectly preserve an identical joint distribution. Regarding the communication transcript, the real incoming message is evaluated as $\boldsymbol{x}_0^{(1)} = \boldsymbol{a}_1 \oplus [\![\boldsymbol{v}]\!]_1$. Since $\boldsymbol{a}_1$ is derived from the matrix expanded via the PRG G, it is computationally indistinguishable from a uniform random vector; otherwise, one could construct an adversary to violate the security of the PRG. Consequently, $\boldsymbol{a}_1$ acts as a computational one-time pad, rendering the real transcript $\boldsymbol{x}_0^{(1)}$ computationally indistinguishable from the uniform random vector sampled by the simulator. Finally, by programming the random oracle as $H(k_w) := [\![\tilde{\boldsymbol{v}}]\!]_0 \oplus \boldsymbol{b}_0$ upon intercepting k_w (where the prior query probability is negligible), the terminal local output matches the target ideal share $[\![\tilde{\boldsymbol{v}}]\!]_0$ perfectly. Therefore, the entire simulated view is computationally indistinguishable from the real world, completing Case 1.

Case 2: Corrupted Server S_1. In the case where S_1 is corrupted, the simulated view generated by $\mathcal{S}_{\mathsf{perm\text{-}off},1}(1^\lambda, [\![\boldsymbol{v}]\!]_1, [\![\tilde{\boldsymbol{v}}]\!]_1)$ is computationally indistinguishable from the real execution view. The specific simulation mechanics and reduction arguments proceed analogously to Case 1 of Lemma 6 and Case 2 of Lemma 5. □

Proof of Theorem 1. For any PPT adversary $\mathcal{A}$ who statically corrupts S_t, we show a PPT simulator $\mathcal{S}$ to simulate the view for $\mathcal{A}$. Then we prove that the simulated view is indistinguishable from real-protocol execution, thus completing the proof.

Simulation of setup. For the setup protocol, the simulator works as follows:

$\mathcal{S}(1^\lambda, t, |\mathsf{DB}|)$:

1: For each block in the DORAM storage, uniformly sample two random nonces η_0, η_1 and ciphertext c as $\eta_0 \xleftarrow{\$} \{0,1\}^\lambda, \eta_1 \xleftarrow{\$} \{0,1\}^\lambda, c \xleftarrow{\$} \{0,1\}^B$. Denote the simulated encrypted database as EDB.
2: Return EDB.

The simulated EDB is indistinguishable from real-world execution due to the security of F. First, η_0 and η_1 are identically sampled as the real protocol execution, so the simulated nonces are identically distributed as the real protocol exeution. The only difference is from c. In the real protocol,

$$c = b \oplus F(k_0, \eta_0) \oplus F(k_1, \eta_1)$$

for a plaintext block b, while $c \xleftarrow{\$} \{0,1\}^B$ follow uniform distribution over $\{0,1\}^B$. Since F is a secure PRF and k_{1-t} is hidden from $\mathcal{A}$. Then $b \oplus F(k_{1-t}, \eta_{1-t})$ is computationally indistinguishable from the uniform distribution over $\{0,1\}^B$; otherwise, we can build a distinguisher to distinguish F from a random function.

Simulation of access without eviction. For the access protocol without eviction, the DORAM protocol involves the Retrieve protocol over a path id. In the ideal world, the simulator works as follows:

$\mathcal{S}(1^\lambda, t, |\mathsf{DB}|)$:

1: Sample a random path id $\xleftarrow{\$} [2^H, 2^{H+1})$.
2: Call the 2-server PIR simulator $\mathcal{S}_{\mathrm{PIR}}$ over the encrypted blocks along the path id.
3: Sample $\eta_0 \xleftarrow{\$} \{0,1\}^\lambda, \eta_1 \xleftarrow{\$} \{0,1\}^\lambda, c \xleftarrow{\$} \{0,1\}^B$, and put (η_0, η_1, c) to the root of the ORAM blocks.

We argue that the above simulation generates a view that is indistinguishable from real-protocol execution. First, note that any block is assigned to a random path id in the real protocol execution, while the simulator also randomly samples a path for simulation. Thus, this part is indistinguishable from real-protocol execution. Second, from the security of PIR, the view from PIR over the blocks along the path id is indistinguishable from real-protocol execution. Overall, the simulation for the Retrieve protocol is indistinguishable from real protocol execution. The last step involves sending the re-encrypted blocks back to the root of the server-side ORAM blocks. The last step of the simulation is indistinguishable from the real-protocol execution from the security of PRF F.

Simulation of eviction. For the eviction protocol in the $\mathcal{F}_{perm}$-hybrid model, the simulator handles the interaction with the ideal functionality and the subsequent write-back phase. The simulator works as follows:

$\mathcal{S}(1^\lambda, t, |\mathsf{DB}|)$:

1: Let $\boldsymbol{v}_t$ be the vector of data blocks along the public eviction path, and convert $\boldsymbol{v}_t$ into secret shares $[\![\boldsymbol{v}]\!]_t$.
2: Call $\mathcal{F}_{\mathsf{perm}}$ with input $[\![\boldsymbol{v}]\!]_t$ to obtain shares $[\![\tilde{\boldsymbol{v}}]\!]_t$, and then call $\mathcal{F}_{\mathsf{perm}}$ simulator $\mathcal{S}_{\mathsf{perm},t}$.
3: Sample a random nonce seed $\mathsf{s}\eta_{1-t} \leftarrow \{0,1\}^\lambda$ and a uniformly random vector $[\![\tilde{\boldsymbol{v}}']\!]_{1-t} \leftarrow (\{0,1\}^B)^{n_s}$ as the incoming transcript.

We argue that the above simulation generates a view that is indistinguishable from the real protocol execution. First, since the simulator tracks the emulated database state, the extracted vector $\boldsymbol{v}_t$ and its corresponding local secret shares $[\![\boldsymbol{v}]\!]_t$ are indistinguishable from the view in the real protocol execution. Second, by the security of the ideal functionality $\mathcal{F}_{\mathsf{perm}}$, the simulation of the shuffle phase introduces zero distinguishable leakage. In the subsequent write-back phase of the real protocol, the honest server S_{1-t} computes its blinded message share via $[\![\tilde{\boldsymbol{v}}']\!]_{1-t}[i] = [\![\tilde{\boldsymbol{v}}]\!]_{1-t}[i] \oplus F(k_{1-t}, \eta_i)$. Because the honest server's PRF key k_{1-t} is sampled uniformly at random

and strictly hidden from the adversary's view, the evaluations of $F(k_{1-t}, \cdot)$ are computationally indistinguishable from truly random strings under the standard PRF assumption. Consequently, these pseudorandom strings serve as a one-time pad that fully masks $[\![\tilde{\boldsymbol{v}}]\!]_{1-t}$, rendering the real vector $[\![\tilde{\boldsymbol{v}}']\!]_{1-t}$ computationally indistinguishable from a uniformly random vector. The simulated transcript $(\mathsf{st}_{1-t}, [\![\tilde{\boldsymbol{v}}']\!]_{1-t})$ is thus computationally indistinguishable from the real messages transmitted over the inter-server channel.

Summary of Overall Indistinguishability. Combining the indistinguishability guarantees of the setup, access without eviction, and eviction phases, the simulator preserves the joint distribution of the entire execution view. Since the simulator continuously and accurately updates the emulated database state across any polynomial sequence of adaptively chosen queries, no PPT adversary can distinguish the real-world execution from the ideal-world simulation with non-negligible advantage. This concludes the proof of Theorem 1.

□

E Detailed Experimental Results

This appendix provides the comprehensive numerical data used to generate the performance benchmarks illustrated in Section 5.2. The following tables present a granular comparison between DuetORAM and existing DORAM schemes, including S^{3}ORAM [24] and DUORAM [38].

Throughout this paper, the suffixes 64B and 128B denote block sizes of 64 Bytes and 128 Bytes, respectively. For timing measurements, we report results in milliseconds (ms) for LAN settings (30μs RTT, 100 Gbps) and seconds (s) or milliseconds (ms) for WAN settings (30ms RTT, 100 Mbps) as specified in each table. Communication costs are reported in Kibibytes (KiB) or Mebibytes (MiB).

Table 3: Detailed Retrieval Latency (ms) for Block Size = 8 Bytes

Schemes	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
LAN Setting						
DuetORAM	1.05	1.08	1.12	1.15	1.22	1.55
S^3ORAM	1.84	1.86	1.90	1.96	2.00	2.09
DUORAM	188.72	190.36	189.48	194.17	318.05	327.16
WAN Setting						
DuetORAM	305.98	306.01	306.02	306.20	306.24	306.31
S^3ORAM	308.41	308.42	308.42	308.42	308.43	308.44
DUORAM	348.85	347.88	350.14	353.32	467.42	467.16

Table 4: Detailed Retrieval Latency (ms) for Different Block Sizes. Here, 64B and 128B denote block sizes of 64 Bytes and 128 Bytes, respectively.

Schemes	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
LAN Setting						
DuetORAM-64B	1.55	1.57	1.61	1.84	2.15	2.33
S^3ORAM-64B	2.12	2.16	2.48	2.67	2.79	3.09
DuetORAM-128B	1.84	1.98	2.11	2.31	2.72	3.17
S^3ORAM-128B	2.48	2.77	2.99	3.28	3.61	3.96
WAN Setting						
DuetORAM-64B	306.82	307.04	307.04	307.07	307.06	307.09
S^3ORAM-64B	309.81	309.74	309.85	310.00	310.20	310.42
DuetORAM-128B	307.79	307.86	308.39	309.03	308.41	308.37
S^3ORAM-128B	310.93	311.19	311.26	311.40	311.95	312.11

Table 5: Detailed Eviction Latency Comparison (s)

Schemes	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
LAN Setting						
DuetORAM-Off-64B	1.09	1.23	1.19	1.46	1.72	1.89
DuetORAM-On-64B	0.21	0.24	0.28	0.31	0.36	0.39
S^3ORAM-64B	1.73	1.78	1.90	2.07	2.33	2.60
DuetORAM-Off-128B	1.51	1.90	2.16	2.68	3.01	3.57
DuetORAM-On-128B	0.25	0.25	0.26	0.29	0.31	0.37
S^3ORAM-128B	1.54	1.76	1.97	2.12	2.27	2.52
WAN Setting						
DuetORAM-Off-64B	1.91	2.20	2.52	2.71	2.83	3.22
DuetORAM-On-64B	1.54	1.75	1.98	2.10	2.23	2.27
S^3ORAM-64B	7.82	7.92	9.77	10.41	10.81	11.57
DuetORAM-Off-128B	2.93	3.03	3.34	3.88	4.22	4.92
DuetORAM-On-128B	1.92	2.02	2.22	2.30	2.36	2.43
S^3ORAM-128B	7.09	8.38	9.79	10.01	10.85	11.77

Table 6: Detailed Client-Server Communication Cost per Retrieval (KiB)

Schemes	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
DuetORAM-64B	3.12	3.45	3.77	4.10	4.42	4.75
S^3ORAM-64B	23.55	26.16	28.76	31.36	33.96	36.56
DuetORAM-128B	3.25	3.57	3.90	4.22	4.55	4.87
S^3ORAM-128B	23.68	26.28	28.88	31.48	34.09	36.69

Table 7: Detailed Client-Server Communication Cost per Eviction (MiB)

Schemes	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
DuetORAM-Off-64B	0.076	0.084	0.091	0.099	0.107	0.114
DuetORAM-On-64B	0.635	0.699	0.762	0.826	0.889	0.953
S^3ORAM-64B	15.228	16.920	18.612	20.304	21.996	23.688
DuetORAM-Off-128B	0.076	0.084	0.091	0.099	0.107	0.114
DuetORAM-On-128B	1.245	1.369	1.494	1.618	1.743	1.867
S^3ORAM-128B	15.228	16.920	18.612	20.304	21.996	23.688

Table 8: Detailed Inter-Server Communication Cost per Eviction (MiB)

Schemes	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
DuetORAM-64B	0.81	0.89	0.98	1.06	1.14	1.22
S^3ORAM-64B	1.46	1.63	1.79	1.95	2.11	2.28
DuetORAM-128B	1.63	1.79	1.95	2.11	2.28	2.44
S^3ORAM-128B	2.93	3.25	3.58	3.90	4.23	4.55