V-ORAM: A Versatile and Adaptive ORAM Framework with Service Transformation for Dynamic Workloads

Bo Zhang[†], Helei Cui[†(✉)], Xingliang Yuan[◇], Zhiwen Yu[†‡], and Bin Guo[†]
[†]*School of Computer Science, Northwestern Polytechnical University, China*
[◇]*School of Computing and Information Systems, The University of Melbourne, Australia*
[‡]*College of Computer Science and Technology, Harbin Engineering University, China*
[✉]*Corresponding author, email: chl@nwpu.edu.cn*

Abstract

Oblivious RAM (ORAM) has been attracting significant attention for building encrypted data storage systems due to its strong security guarantees and communities' continuing efforts to improve its efficiency. Despite great potential, a specific ORAM scheme is normally designed and optimized for a certain type of client workloads, given the nature of its complicated cryptographic construction. Once deployed, a single ORAM service can hardly serve dynamic workloads in an efficient and cost-effective manner.

To bridge the gap, in this paper, we propose a versatile ORAM framework named V-ORAM, which can efficiently and securely switch between different ORAM services to adaptively serve dynamic workloads in the real-world. In particular, V-ORAM is equipped with a service transformation protocol that leverages a base ORAM as an intermedia of transformation and can synchronize the states of tree-based ORAMs without downloading and rebuilding the ORAM by the client. We formalize the security of V-ORAM, and prove that V-ORAM holds the security of ORAMs, including the process of service transformation. V-ORAM also provides a planner to recommend the ORAM service type and ORAM parameters for adapting to the client workloads, server resources and monetary expenses. We implement V-ORAM and evaluate the cost of transformation. We also conduct real-world case studies over three medical datasets and different workloads. Compared with directly rebuilding ORAMs, V-ORAM saves up to $10^{4.12}\times$ processing time and communication cost, up to 33.1% of monetary costs in real-world workloads, and generates constant impact to employed ORAM services, *i.e.*, $<$ 5ms in processing and $<$ 50kB in communication.

1 Introduction

Encrypted data storage systems enable a client (*i.e.*, data owner) to securely outsource their sensitive data to the cloud server while retaining normal data access and query functionalities [26, 34, 53, 54]. Those systems can be designed via various cryptographic tools with diverse levels of functionality, performance, and security. Among others, systems [31, 50, 59] built from Oblivious RAM (ORAM) [42] are attracting much attention, given its strong security features and the increasing effort in improving its performance in the literature. For example, E2EE application Signal [12] leverages ORAM to protect customers' phone numbers, and Cloudflare provides ORAM-based encrypted file storage service named UtahFS [15].

In essence, ORAM is a cryptographic data structure that can ensure the client's data confidentiality against the server while concealing the client's access pattern to the server [26, 58, 64]. Namely, the server can neither learn any data content of the client nor whether the client has accessed the same data before. Due to randomized access patterns, ORAM-based systems are resilient to emerging leakage attacks against encrypted data storage systems [24, 52, 78, 81].

Despite its great potential, ORAM is considered heavy due to continuously shuffling and re-encrypting data once being accessed by the client, *i.e.*, with *polylogarithmic* communication and computation costs [42, 64]. Such prohibitive costs also limit its application. To change the status quo, many recent studies have devoted to pushing forward its performance (*e.g.*, involving server computation [30, 36, 58] and leveraging data locality [25, 28, 47]) and broadening its functionality for wider application (*e.g.*, asynchronous networks [26, 32, 59], searchable encryption [50, 77], and secure computation [43, 69]).

However, there remains an unfulfilled gap in practice, *i.e.*, the demand for ***serving dynamic workloads***. As an exemplary scenario, a hospital stores its patient data in a cloud-based Electronic Health Record (EHR) system and accesses the data via a cloud-provided API in routine without special performance requirements. However, remote care (or telemedicine) [14] supported by EHR requires real-time data or video stream processing, and EHR redundancy or backups to prevent catastrophic downtime [3, 9]. Both of these demands necessitate lower latency and higher throughput. On the other hand, EHR typically supports advanced data search [4]. Typically, in routine consultations, doctors search data via simple keywords. Meantime, in data analysis and clinical decision-making [7,65,66], analysts utilize more complex queries like join or aggregate. Other examples include financial transaction systems like stock, which require timely updates of stock prices, often involve multiple clients, and demand higher throughput during peak periods. Similarly, social media applications like Signal [11] experience peak/off-peak as well, normally require multi-client parallel access and privately search phone numbers for contact discovery [13,67].

In plaintext systems, query planners can handle such dynamic workloads well. However, in an encrypted data storage system with an ORAM back-end, adapting to such dynamics presents compelling challenges. A single ORAM service is normally designed and optimized for one specific type of workload (§2.2), so it can hardly accomplish requirements from dynamic workloads. For better performance and cost-effectiveness, clients desire to utilize the matched ORAM service to a particular workload.

What if using unmatched ORAM? Clients who keep using a specific ORAM construction to serve dynamic workloads will: (1) degrade system performance, *e.g.*, using asynchronous ORAM [26,59] in a synchronous network causes unnecessary synchronization, resulting in performance degradation; (2) aggravate monetary cost, *e.g.*, utilizing standard ORAM [64] under workloads with high throughput increases monetary costs to meet performance requirements.

How to serve dynamic workloads? A straightforward solution is to maintain all the client-necessitated ORAMs in the server. Once the client's workload is changed, the system for the client will opt for a corresponding ORAM. However, ORAM is *stateful* [30, 58, 64], synchronizing states of two ORAMs involves a heavy burden on the system. It also multiplies the server-side storage cost and client access costs. The other naive solution is to rebuild the ORAM, *i.e.*, the client downloads the ORAM from the server, re-constructs, and re-uploads the new ORAM. Yet it introduces large communication, computation costs and undesired service interruption.

Our approach. A desired solution is to enable the system to *efficiently transform between different ORAM services*, without either maintaining multiple different ORAM constructions for the same database or invoking rebuilding of the entire encrypted database. In this paper, we propose V-ORAM, a versatile ORAM framework, that can switch between different ORAM services to serve dynamic workloads over the encrypted data storage. In particular, V-ORAM supports three representative tree-based ORAM services *i.e.*, Path ORAM [64], Ring ORAM [58], and ConcurORAM [26], which can sufficiently support five commonly desired workloads in the real-world (§2.2). V-ORAM includes a newly designed ORAM service transformation (OST) protocol. It is tailored to securely and efficiently switch between ORAM services with small client and server costs.

In addition, as ORAM involves complex processes with many tunable parameters, deciding when and which ORAM service to transform to can be tricky for non-expert clients.

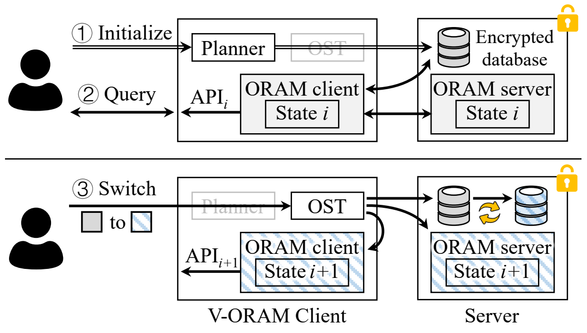

Figure 1: System workflow of V-ORAM. ① Client sends workloads to the planner, initializes, and uploads the encrypted database to the server. ② Client queries V-ORAM through API of current ORAM. ③ To switch ORAM service, OST updates ORAM states on the client and server.

Thus, V-ORAM further offers a ***planner*** with systematic modeling linking the monetary cost to ORAM performance. The planner provides the client with a decision strategy, *e.g.*, ORAM types suitable to its workloads, optimal ORAM parameters based on its datasets, and server resource configurations.

We implement a prototype of V-ORAM including three aforementioned ORAMs and conduct comprehensive evaluations. Experimental results show that, compared with rebuilding the ORAMs, V-ORAM reduces up to $10^{4.12}\times$ of processing time and communication, and only generates extra amortized process time < 5ms and communication cost < 50kB on ORAM services. We further evaluate V-ORAM in real-world with the block I/O trace of server *src*1_0 in MSRC datasets [51]. V-ORAM saves up to 33.1% monetary costs while adhering to the performance requirements.

1.1 Summary of Techniques

V-ORAM considers a classic system setting that a client outsources its data to a server [26,58,59,64]. Figure 1 illustrates the workflow of V-ORAM. During initialization, the client generates the encrypted database from their dataset. Then, the client sends it to the server and builds the ORAM server backend. We abstract ORAM services into an ORAM client with black-boxed APIs and states. To access the data, the client invokes APIs and retrieves encrypted data from the server.

In V-ORAM, OST protocol handles service switching requests and updates ORAM states. Naturally, OST should adhere to the following design goals. (1) Switches between ORAMs correctly without data loss or overflow. (2) Safeguards the security of the overall V-ORAM, including confidentiality and data obliviousness. (3) Enables efficient transformation whose costs do not scale with the database size. We stress that achieving the above goals is non-trivial due to several technical challenges which will be elaborated below.
Challenge #1: Finding a transformation intermedia. De-

signing a universal transformation protocol for any arbitrary two ORAMs appears technically infeasible and error-prone due to diverse ORAM constructions. OST circumvents this by designating a specific ORAM as the intermediary, as we termed *base ORAM*. To switch to a desired ORAM, OST first switches the current ORAM to the base ORAM and then switches to the target ORAM. This allows us to only design a single set of transformation protocols for each ORAM. Therefore, we need to identify a base ORAM that (1) possesses compatible construction with other ORAMs and (2) has the potential for efficient transformation.

Our solution (§4.1). We analyze the considered ORAMs in-depth and estimate the costs associated with different ORAMs as the base ORAM. We figure out that Ring ORAM [58] satisfies our demands. It maintains the metadata required by other considered ORAM [26], which scales linearly with the database size and avoids extensive data transition. Its query procedures are also compatible with other ORAMs. Upon Ring ORAM, we design transformation protocols with constant costs, which is non-trivial as two leakages are caused by inherent features of Ring ORAM as follows.

Challenge #2: Handling the ORAM states. ORAMs are stateful [26, 58, 64], preserving metadata essential for their functionality. These states are vital for the correctness, security, and performance of ORAMs. We observe that abrupt transformation that eliminates previous states leads to severe leakages that compromise the security of ORAMs. The leakages arise when the query *re-accesses invalid blocks* or *exposes the block positions* due to the transformation. Thus, OST should inherit and synchronize the states during the transformation.

Our solution (§4.2). We design client-side metadata called ***record map***, which logs the status of blocks and helps to refresh the blocks before any leakage (*i.e.*, EvictRecord). Our transformation protocol designed upon them enables secure service transformation and is theoretically and empirically proven to have constant costs.

Challenge #3: Formulating a decision strategy. We envision a strategy taking the client workloads, performance constraints, and monetary expense as input to determine the sequence of ORAM services and their essential parameters that fulfill the requirements. Thus, V-ORAM expects to systematically model the ORAM from monetary cost to its performance and be applicable to complex real-world scenarios.

Our solution (§5). We construct a systematic model grounded in the query procedures of ORAM. It estimates the ORAM performance according to the computation, communication, and storage capabilities of the server. Then we estimate the monetary costs through the server capabilities according to the pricing provided by the cloud.

Contributions. To the best of our knowledge, we are the initial exploration into dynamic workloads on ORAM. The primary contributions of our work are outlined as follows:

- We introduce the V-ORAM framework, enabling efficient transformations between ORAMs.
- We propose the OST protocol, mitigating the leakages caused by abrupt transformation with constant costs. We also offer a planner to assist client decision-making.
- We implement a prototype of V-ORAM and conduct comprehensive evaluations. Our code is open-sourced[1].

Limitations. (1) V-ORAM currently can only switch between tree-based ORAMs. After thoroughly analyzing workloads and known ORAM schemes, our observation is that the selected tree-based ORAMs already effectively facilitate the most common workloads. We leave transformation among other ORAM schemes as a future direction for more diverse and specialized workloads. (2) The workloads in this paper tend to be periodic, *i.e.*, the clients primarily request a certain service over a period. (3) Our transformation protocol still incurs service interruptions. Addressing concurrent workloads is also designated as future work. Further discussion can be referred to Appendix A.4.

2 Background

2.1 Taxonomy of ORAMs

We now describe the basic workflow of three considered ORAMs. Refer to Appendix A.1 for more detailed protocols.

Path ORAM [64]. In Path ORAM, the client encrypts the data blocks of size B, constructs the bucket with Z blocks, organizes all buckets as a complete binary tree, and outsources the tree to the server. Each bucket is randomly mapped to a path (*i.e.*, the path from a leaf to the root) and is maintained in a client-side metadata position map PosMap. The client also maintains a temporary cache Stash.

Access. The client performs Query and Eviction to access a block. In Query , the client retrieves the entire path from the server and decrypts the blocks in the path to get the required block. The retrieved blocks are then *randomly re-encrypted* and added in Stash waiting for eviction. After the query, the target block is always re-mapped to a random path. Then, the client invokes Eviction to write the blocks in Stash back to the tree. Apparently, the costs of Path ORAM is $O(\log N)$.

Ring ORAM [58] leverages XOR technique to achieve constant online communication costs. Meanwhile, to maintain the constant costs, its eviction is performed *periodically* [58].

Dummy blocks. Ring ORAM incorporates S client reproducible dummies within buckets, and tracks their positions and validity. The client queries a certain block XORed with dummies and decrypts via re-generated dummies. Used dummies are marked as invalid. Once invalid dummies are accessed, the access pattern leaks [58].

ConcurORAM [26] is a concurrent ORAM that allows multiple clients to access the ORAM in parallel without any trusted proxy or complex interaction among clients. It is built upon Ring ORAM with batched requests, which contain at most c

[1] Our code will be available for artifact evaluation if the paper is accepted.

Table 1: Taxonomy of our considered workloads and ORAM schemes.

Workload	Real-world Examples	ORAM Schemes	Orthogonal optimizations				
			Recursion	Concurrency	Partitioning	Construction	Hardware
General Storage	Long-term storage [7], clinical data collection [6].	Path ORAM [64]	✓				
		rORAM [25]					✓
Real-time Updates	Clinical decision [65], big data analytics [21].	Ring ORAM [58]				✓	
		ConcurORAM [26]		✓		✓	
Parallel Access	Health information exchange [8, 66], federated analysis [40, 48].	Snoopy [34]			✓		✓
		ConcurORAM		✓		✓	
		TaoStore [59]		✓	✓		
Resource Constrained	Mobile Apps [20, 76], wearable devices [46].	FutORAMa [18]	✓			✓	
		SCSL11 [61]	✓				
		Path ORAM (recursive)	✓				
Specialized Functionality	Keyword searching [35, 50], ciphertext computation [40, 69, 72, 73], asynchronous network [26, 59].	OBI [77]				✓	
		DUORAM [69]				✓	
		ConcurORAM [26]		✓		✓	

queries. It maintains synchronization metadata via server-side oblivious data structures and mutex to enable parallelism.

Synchronization metadata. There are three primary metadata, *i.e.*, StashSet, PosMap, and DRLogSet. Stashset includes c entries called TempStash, each is encrypted from Stash and contains at most $MaxStashSize$ real blocks and at least c dummies [26]. PosMap is stored on the server in the form of PD-ORAM [75] (a hierarchical asynchronous ORAM). DRLogSet has c entries called data record log (DRL), each records the resultant blocks of the request batch and is encrypted with c real blocks and c dummy blocks.

Eviction types. We classify ORAM eviction into *instant eviction* [59, 64, 72] and *periodic eviction* [30, 36, 58]. Instant eviction is executed for each query and accesses the entire bucket, *e.g.*, Path ORAM. Periodic eviction is triggered periodically (normally every A accesses) and leverages dummy blocks for block-level access, *e.g.*, Ring ORAM and ConcurORAM.

In V-ORAM, we want the dummies to be efficiently consumed to avoid transiting massive dummies in transformations. We also hope these dummies do not exacerbate the cost of instant evictions. Achieving both is non-trivial as we report two leakages caused by abrupt transformation in §4.2.1.

Optimizations for ORAM. A substantial body of studies aim at enhancing performance and enriching application scenarios of ORAM [26, 34, 58, 59, 64]. We divide those studies into five orthogonal optimizations according to their techniques.

- Recursion [39, 64]: Recall the PosMap maintained locally on the client, its size is $O(N)$. Devices with constrained storage may not be able to store the entire PosMap for large datasets. Therefore, a technique is to recursively store PosMap as ORAM on the server, which can reduce the storage overhead to $O(\log N) \cdot \omega(1)$ [64].
- Concurrency [26, 32, 59]: This ORAM variant manages concurrent requests from multiple clients. A typical technique is to obliviously maintain synchronization metadata, guaranteeing the correctness of parallel accesses. ORAMs may keep the metadata on either the client or server side, which we term as client- and server-state (Figure 1).
- Partitioning [59, 62, 63]: Large-scale data result in a binary tree of significant depth, which causes IO and communication bottlenecks. By partitioning, the binary tree is divided into subtrees of a smaller depth. These subtrees are distributed among multiple servers/instances to allow parallel access and alleviate bottlenecks.
- Construction [30, 36, 58, 77]: Some ORAMs require specific block or bucket constructions, as they leverage certain cryptographic primitives to trade computation for $O(1)$ online communication costs, *e.g.*, XOR, private information retrieval (PIR), homomorphic encryption (HE).
- Hardware [23, 34, 47, 55, 57, 62]: Extensive efforts focus on optimizing hardware I/O by leveraging temporal and spatial data locality. On the other hand, trusted execution environments (TEE) are also leveraged to reduce heavy online costs between the client and server.

Our scope. (1) As we mentioned before, transformations between non-tree-based ORAM schemes are considered out of scope. (2) Among these optimizations, our primary challenge is to securely and effectively transform between selected ORAMs, due to their different eviction procedures and intricate synchronization. Other optimizations can be naturally inherited if V-ORAM supports the corresponding ORAMs, *i.e.*, recursion, partitioning, and hardware. (3) V-ORAM also accounts for ORAMs with different ciphertext constructions. Further discussions can be referred to Appendix A.4.

2.2 Workloads in Practice

We take cloud-based EHR, prominently referred to in oblivious security [33, 34, 44], as an example. We merge real-world products with the academic sphere and point out commonly desired workloads. Table 1 illustrates the relationship between the representative ORAM schemes and workloads, alongside the aforementioned optimizations. Refer to Appendix A.2 for a more detailed description of workloads.

Why we choose these ORAMs? The selected ORAM schemes cover the three of our considered workloads [26, 58, 59, 64]. Concretely, Path ORAM represents the general storage, Ring ORAM serves for real-time updates and the base ORAM of transformation, ConcurORAM is suitable for parallel access. As for ORAMs designed for left two workloads, they are readily compatible with V-ORAM, *e.g.*, resource-constrained devices can directly store PosMap in smaller ORAMs, specialized functionality like SSE-oriented ORAMs can replace blocks with encrypted search indexes.

3 System Overview

3.1 System Model and Threat Assumptions

As shown in Figure 1, V-ORAM includes a client and a server.

Client. In initialization, the client first determines ORAM parameters and service sequence through the planner based on its datasets and workloads. The client then generates the encrypted database, outsources it to the server, and accesses the data via APIs of the current ORAM service. To switch to the predetermined ORAM service, the client invokes OST APIs to transform the client- and server-side ORAM state.

Server. The server stores client data and interacts with the client through APIs. We assume the server has enough memory to accommodate all metadata and store the main body of the data in storage (SSD). The server configuration is assumed to be elastic and can be modified by the client.

Threat assumptions. Our threat model includes an honest client and a semi-honest server. The client does not maliciously poison data or tamper with metadata. The server honestly executes procedures without modifying or discarding client data or requests. Communication channels of V-ORAM are assumed to be authenticated and encrypted using TLS.

Threats during service transformation. (1) Adversary is assumed to be capable of recording historical accesses and tracking the access status of blocks across ORAM services. (2) Adversary can monitor communications throughout the V-ORAM. While the specific content is protected by TLS, the adversary can record the size of requests. (3) APIs and their invocations are assumed public. The adversary can observe API calls (including OST APIs) and identify the current service type, but the content of the invocations is protected. (4) The adversary is assumed to be aware of the size and historical accesses of server-side metadata, but cannot learn the specific content. (5) The adversary cannot learn any internal client operations, e.g., read/write, encryption/decryption, and only observe externally encrypted requests. (6) The adversary cannot learn any client secrets, e.g., encryption keys, PosMap, Stash, and other metadata. Our threat model is also aligned with existing research [26, 34, 58, 59, 64].

Other threats against ORAMs. We assume that the client could securely access the current ORAM service via its API. Furthermore, we assume the timing attacks in asynchronous networks [34, 59] can be defended by the employed concurrent ORAM. The attacks that can not be defended by employed ORAMs are regarded as out of our scope.

3.2 Security Formalization

This section presents the security definitions and theorems. We formally prove the security of our system in Appendix B.1.

In V-ORAM, the adversary could learn public information like access of each query, data size (block and bucket size), and server-side I/O. In this regard, V-ORAM protects private information including data content and client access patterns. We stress that V-ORAM *does not protect the service types of ORAM*, as this is a system parameter and does not appear to be related to data content.

Thus, we split the security of V-ORAM into two parts: (1) the obliviousness of data access with our proposed eviction algorithm, and (2) the security of the overall system.

Obliviousness. Since V-ORAM offers different ORAM services, we redefine the access sequence as follows.

$\vec{y} = \{(\mathsf{op}_1, \mathsf{add}_1, \mathsf{data}_1, \mathsf{sid}_1), \ldots, (\mathsf{op}_m, \mathsf{add}_m, \mathsf{data}_m, \mathsf{sid}_k)\}$ denotes an access sequence with length of m and k service types, where $\mathsf{op}_i \in \{\mathsf{read}, \mathsf{write}\}$ denotes the operation type, add_i denotes the address of accessed data, data_i denotes the content being written, and sid_j denotes the service type.

Following the existing definitions of ORAM [58, 64], the obliviousness of V-ORAM can be refined as follows.

Definition 1 (Data Obliviousness)**.** *For a given access sequence $\vec{y}$ of length m, and a random sequence $\vec{z}$ with the same length and identical service types, V-ORAM is oblivious if for any probabilistic polynomial-time (PPT) adversary* Adv*, the advantage of* Adv *distinguishing $\vec{y}$ from $\vec{z}$ is $negl(m)$.*

As we assumed, accesses within any ORAM service period satisfies the above definition. In other words, for a continuous access sequence with the same sid_j, the client can securely access ORAM$_j$ via the corresponding API.

System security. To prove the security of overall framework, we first simulate the functions of V-ORAM solely based on public information. Our construction is secure as long as any PPT Adv cannot distinguish whether it is interacting with a real V-ORAM (*i.e.*, "real world") or an ideal simulation that is limited to public information (*i.e.*, "ideal world"). We present the informal definition of the experiments in the following, with the formal details available in Appendix B.1.

Real and ideal world (informal). In the real world, the client initializes V-ORAM according to Figure 3 and invokes the APIs to access data. To switch the services, the client performs the protocol illustrated in §4.2. Adv can observe all client requests and server-side I/O. The ideal world is the same as the real world, except all the data is generated by random simulators based on public information. Adv can observe the simulator-generated traces as well. Adv's goal is to distinguish between the real world and the ideal world.

We denote the protocol of real-world as Π and the ideal world as Sim. The security of V-ORAM is further defined as:

Definition 2 (System Security)**.** *The V-ORAM framework Π is secure if for any given access sequence $\vec{y}$ and PPT adversary Adv, there exists a PPT simulator Sim that*

$$\left|\Pr\left[\mathbf{Real}^{\Pi}_{\mathsf{Adv}}(\lambda)=1\right]-\Pr\left[\mathbf{Ideal}^{\mathsf{Sim}}_{\mathsf{Adv}}(\lambda)=1\right]\right|\leq negl(\lambda)$$

where λ denotes the security parameter of V-ORAM.

To be compatible with obliviousness, the simulations generate random access sequence $\vec{z}$ with the same length and service types. We defer the detailed simulations in Appendix B.1.

4 Design of V-ORAM

In this section, we introduce how we choose the base ORAM and perform ORAM service transformations, based on the V-ORAM framework in Figure 1. We first explain the selection of base ORAM, followed by an introduction to our OST protocol, including the aforementioned leakages and our mitigation, $\mathsf{EvictRecord}$. Lastly, we wrap up and present the overall workflow of V-ORAM.

4.1 Base ORAM

We again stress that it is infeasible to design transformation protocols for arbitrary two ORAMs due to their various constructions. Instead, we select a base ORAM as the intermedia of transformation, which is required to be:

- Compatible: The base ORAM ought to be tree-based and its data/metadata construction should be the most primitive version (*e.g.*, plaintext client-side metadata), thereby can be easily transformed into other constructions.
- Affordable: As the server resource should meet the minimal requirements of base ORAM, we want base ORAM to be affordable without heavy computation or communication.
- Transformation-efficient: The base ORAM should contain the necessary data of other ORAMs, to avoid frequent and massive data transitions during the transformations.

We investigate the mainstream ORAMs and determine that Ring ORAM is the best candidate for base ORAM. The reasons stand for (1) Ring ORAM shares a tree structure and has a compatible access protocol to other two ORAMs (later discussed in §4.2.1). (2) Ring ORAM is among the most efficient schemes in practice with $O(1)$ online costs and $O(\log N)$ amortized costs [58], attracting research continuously improving its performance [23,55,56]. More importantly, it achieves such efficiency via XOR rather than complex cryptographic tools. (3) Ring ORAM preserves all the data required by other ORAMs on the server already, *i.e.*, dummy blocks.

Based on Ring ORAM, we design efficient transformation protocols with costs depending only on the security parameters and ORAM-specified constants (see Table 2 of §6.2).

4.2 Service Transformation Overview

At a high level, service transformation should ensure that the current metadata aligns with the ORAM service. Once the metadata is securely transformed, V-ORAM can replace its API with the new ORAM's API and initiate the new service.

Since Ring ORAM serves as the base ORAM, here we discuss the transformations of ConcurORAM and Path ORAM.

***Transformation of ConcurORAM*.** Recalling §2.1, ConcurORAM maintains metadata on both sides for synchronization, which is initialized in each request batch. Therefore, it is wise to perform transformations after an entire batch, as this avoids dealing with complex mutex. Therefore, when transforming from ConcurORAM, we pad the requests into a full batch.

Like our base ORAM, ConcurORAM employs periodic eviction leveraging dummy blocks and maintains metadata to record the valid state of dummies. This metadata should be correctly inherited by subsequent ORAMs. Otherwise, it will result in re-accessing the invalid dummies and lead to extra leakage, as shown later in the next section.

Transformation of Path ORAM. Recall that Path ORAM only maintains Stash and PosMap in client-side, which can be directly inherited. However, before two ORAMs directly replace the API, an issue arises: which blocks should the Path ORAM access? This stems from that Path ORAM does not leverage dummies. If Path ORAM retrieves all $Z+S$ blocks from the bucket, it doubles the access costs, as $S \approx Z$ [58]. Therefore, we prefer Path ORAM to access its original Z blocks excluding dummies, which also minimizes protocol modifications and improves performance.

Unfortunately, this approach reveals the positions of dummies after Path ORAM's accesses. It does not affect Path ORAM directly but once V-ORAM switches to ORAMs with periodic eviction and accesses these dummies for XOR, the server could infer the positions of the accessed real blocks. This leads to the second leakage discussed in the next section.

4.2.1 Record Map and EvictRecord

***Leakage Analysis from Naive Approaches*.** As illustrated in Figure 2, each bucket contains real and dummy blocks. The blocks in actual ORAM are randomly shuffled, here we arrange them for simplicity. The client can only query the valid block marked as "T" in metadata. Once a valid dummy is read, its metadata is set to "F". The above transformation causes leakages in the following two cases:

- Re-accessing invalid dummy: Figure 2(**Up**) illustrates the leakage when switching between ORAMs with periodic eviction. Taking switching from Ring ORAM to ConcurORAM as an example, metadata elimination makes the invalid dummies "valid" again, *i.e.*, dummies 2 and 3. If ConcurORAM reads these dummy blocks, *i.e.*, "Read dummy 2", the dummy blocks can be accessed twice. This breaks the secure requirements of Ring ORAM [58] (§2.1). The

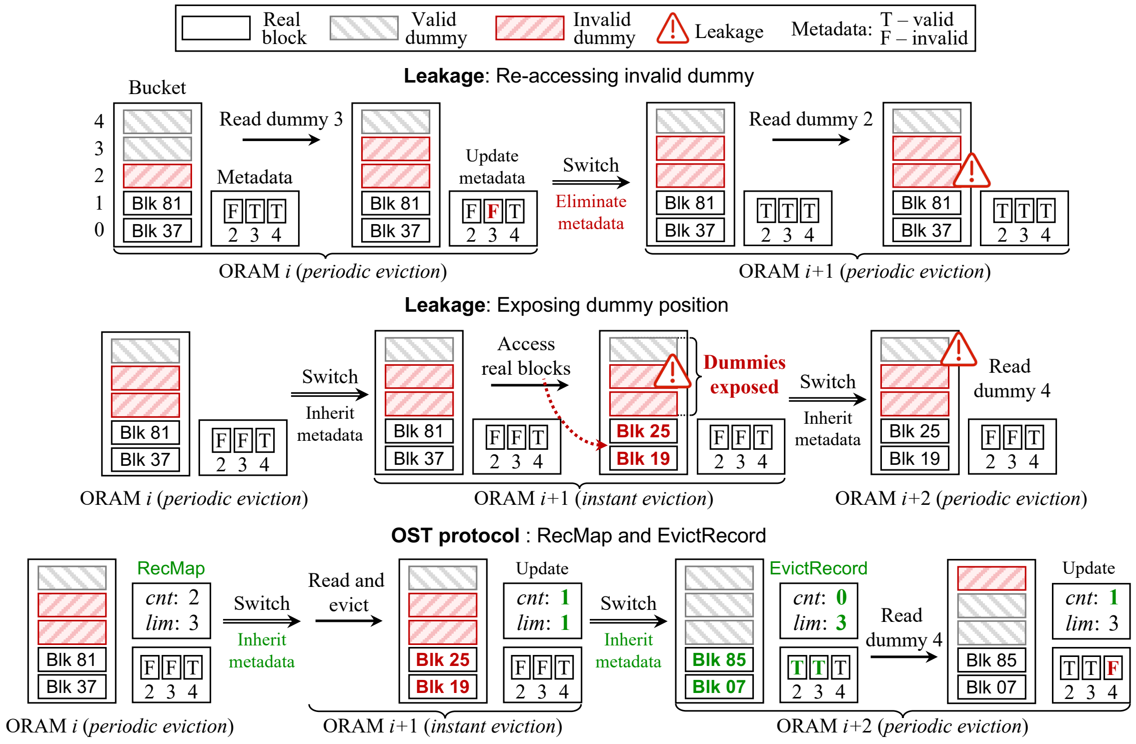

Figure 2: Toy examples of two leakages and our mitigation. (**Up**) Metadata elimination makes the subsequent ORAM unaware of and re-access previously invalid dummies. (**Middle**) Instant eviction only accesses real blocks in buckets, exposing the dummy position to the server. (**Down**) Record map tracks the status of buckets and evicts the invalid dummy via EvictRecord.

server distinguishes these re-accessed blocks as dummy blocks and reveals the access pattern of certain blocks over time. Thus, *the metadata of dummies must be inherited.*

- Exposing dummy position: Figure 2(**Middle**) describes the leakage during switching between ORAM with instant eviction. Take Path ORAM for instance, as we just mentioned, Path ORAM only reads and evicts the real blocks, in which server learns that the remaining blocks are dummies. Once any following periodic evictions re-access these dummies, the accessed real block could be leaked. Such leakage happens even if the metadata is inherited, *i.e.*, "Read dummy 4" in Figure 2(**Middle**). Thus, *the buckets accessed by instant eviction must be evicted before periodic eviction.*

Solution. Regarding the first leakage, OST must inherit the metadata dummy blocks across ORAMs, *i.e.*, their positions, validity, and access status. In this way, the ORAMs with periodic eviction can correctly identify the valid dummies and avoid re-access. As for the second leakage, OST must record the buckets accessed by instant eviction and evict these buckets before they are accessed by periodic evictions.

We omit the details of the former two metadata in the following as they are adoptable from Ring ORAM [58].

Record map. In Ring ORAM, one bucket can be accessed at most S (the number of dummy blocks) times [58]. We generalize this concept and define the maximum access that a bucket can be securely accessed as ***access limit***, denoted as lim, *i.e.*, the following invariant when selecting lim:

Invariant. Access limit lim_j of ORAM$_j$ ensures the buckets can be securely accessed by API.Query$_j$ at most lim_j times.

Any bucket reaching its limit will be evicted, *e.g.*, Path ORAM's access limit is 1, as the buckets are immediately evicted after accesses. Inspired by EarlyReshuffle of Ring ORAM [58], we design a data structure termed as record map RecMap[bid] $= (cnt, lim)$ to track the access status of buckets, where bid denotes the bucket identifier, cnt denotes its access count. During queries, cnt's of the buckets are increased by 1 if they are read by API.Query, and are reset to 0 if they are evicted. Then, we introduce the EvictRecord algorithm.

EvictRecord. As Algorithm 1 illustrates, we denote read_bids and evicted_bids as the buckets to be read/evicted during ORAM accesses, which are predetermined by ORAM from client-side metadata [26,58,64]. Firstly, all entries in RecMap are initialized with $(0, lim_b)$, where lim_b denotes the access limit of base ORAM. Then, there are two phases in EvictRecord, before query and after eviction, preventing the second and first leakage respectively. (1) Before API.Query, OST evicts the buckets (in read_bids) accessed by previous instant ORAMs. Otherwise, the query could leak the accessed real block to the server. (2) After API.Eviction, buckets reaching their limits are evicted. Thereby avoiding the afterward

Algorithm 1: EvictRecord procedures.

```
Input: read_bids/evicted_bids denote the buckets to
       be read/evicted in this query, current service
       sid, current access limit $lim$.
1  for $i \leftarrow 0$ to $N-1$ do                 // Initialization
2      RecMap[bid$_i$] = $(0, lim_b)$;
3  for bid $\in$ read_bids do              // Before API.Query
4      $cnt, lim'$ = RecMap[bid];
5      if $cnt = lim'$ and periodic eviction then
6          Read blocks in bid to Stash;
7          Write back blocks from Stash;
8          RecMap[bid] = $(0, lim)$;
9  for bid $\in$ read_bids do
10     $cnt, lim' \leftarrow$ RecMap[bid];
11     RecMap[bid] = $(cnt+1, \min(lim, lim'))$;
12 for bid $\in$ evicted_bids do          // After API.Eviction
13     if periodic eviction then RecMap[bid] = $(0, lim)$;
14     else RecMap[bid] = $(1, 1)$;
15 for bid $\in$ evicted_bids do
16     $cnt, lim'$ = RecMap[bid];
17     if $cnt = lim'$ and periodic eviction then
18         Evict bid (Line 6–7);
19         RecMap[bid] = $(0, lim)$;
```

periodic eviction of illegally accessing any dummies.

Note that instant eviction only updates the RecMap as their accesses are immediately followed with evictions (Line 5&13&17). We also place EvictRecord after API.Eviction to avoid repeated evictions. We stress that in Line 11, lim is updated to a smaller limit of the previous and the current limit. Otherwise, the bucket could still be accessed exceeding the previous smaller lim. We formally analyze its security in Appendix B.1 and further theoretically prove its constant amortized costs in §4.4.

4.2.2 OST Protocol

We first explain how OST safeguards security through the above toy example and describe the general flow of OST.

In Figure 2(**Down**), without losing generality, we assume ORAM i and $i+2$ to be Ring ORAM, ORAM $i+1$ to be Path ORAM. After switching to Path ORAM, RecMap of the bucket is set to $(1,1)$, but does not trigger EvictRecord. Then, V-ORAM switches to Ring ORAM, once the bucket is queried, EvictRecord is triggered to reshuffle the bucket. All the dummies in the bucket are replaced with new valid dummies (thus the metadata of three dummies is set to "T"), and RecMap of the bucket is reset to $(0,3)$. After that, Ring ORAM can securely query the blocks in the bucket, *e.g.*, "Read dummy 4" reads the dummy that just refreshed by EvictRecord and updates the RecMap to $(1,3)$.

We then go through the workflow of OST shown in the right column of Figure 3.

Transformation of ConcurORAM. When switching to ConcurORAM, OST first encrypts and pads the Stash to TempStash of size $MaxStashSize + c$. TempStash is then uploaded to the server (recall §2.1). Regarding PosMap, we maintain a server-side PosMap in PD-ORAM, in line with ConcurORAM [26]. The client updates the PosMap of both sides during the query, generating additional $O(\log N)$ costs with a much smaller hidden constant. Then, the client inherits the metadata and RecMap, and then replaces the API.

When switching back from ConcurORAM, the client first finishes the padded requests, as we discussed before. Then, the client retrieves DRLogSet and TempStash to recover Stash. Note that OST needs to traverse the entire DRLogSet to avoid missing any blocks. On the contrary, OST only scans the latest TempStash, as traversing the entire StashSet will repeatedly include evicted blocks. After preparing the metadata, the client replaces the APIs and starts the next ORAM service.

Transformation of Path ORAM. Thanks to our EvictRecord protocol, the transformation of Path ORAM is simple and effective. The only thing the client needs to do is to inherit metadata of dummy blocks and RecMap. After that, the client replaces APIs and activates the Path ORAM service.

4.3 Wrapping Up

In previous sections, we introduce the EvictRecord protocol and the OST protocol. As shown in Figure 3, we now wrap up by presenting the overall workflow of V-ORAM. In initialization, the client encodes the data and prepares the metadata and APIs of V-ORAM. After that, the client accesses data via API or switches the ORAM as necessary via OST. Inline with TaoStore [59], each ORAM is described as a pair ORAM = (Encoder, API). Encoder encrypts/decrypts the data and API contains black-boxed interfaces of each ORAM, *e.g.*, Query, Eviction, and Handler.

Initialization. The client first encrypts dataset D into $\hat{D}$, through Encoder of base ORAM. Then the client constructs the data tree and PosMap, and sends them to the server. After sending the data, the client and server then load the API of the base ORAM and start the ORAM service.

Block encryption. Each ORAM service is mapped to a service id sid through PRF, *e.g.*, the sid of Ring ORAM can be defined as PRF('*ring*'). The client secretly maintains a master key mk and generates the encryption key sk through PRF(mk, sid). This allows the client to easily identify and decrypt blocks from multiple ORAM services. We formally prove the security of encryption in Appendix B.1.

Client access. The client invokes the first phase of Evictrecord to reshuffle the buckets accessed by instant eviction. It then invokes API.Query and interacts with the server's API.Handler to access the data. After obtaining the results,

Initialization:

Input Data: Datasets D and ORAM pair (Encoder, API).

1) Client-side:
- Set ORAM parameters and service sequences by planner.
- Encrypt D into $\hat{D}$ via Encoder$_b$ of base ORAM.
- Build data tree and PosMap (in PD-ORAM) with $\hat{D}$.
- Load API.Query$_b$ and API.Eviction$_b$.
- Send data tree, PosMap, Encoder, API to server.

2) Server-side:
- Upon receiving the data:
  - Prepare PosMap and its API, load API.Handler$_b$.

Block Encryption:

Input Data: Client master key mk, sid, PRF, and plaintext data or retrieved block block.

Encoder.Enc:
- Generate secret key $sk \leftarrow \mathsf{PRF}(mk, \mathsf{sid})$.
- Encrypt data into cipher with sk via AES[1].
- Contact sid || cipher as block.
- Return block.

Encoder.Dec:
- Recover sid || cipher ← block.
- Generate secret key $sk \leftarrow \mathsf{PRF}(mk, \mathsf{sid})$.
- Decrypt cipher into data with sk via AES.
- Return data.

Client Access:

Input Data: Operation type op, address of operated data add and data to be written data*.

1) Client-side:
- Invoke OST.EvictRecord[2], API.Query(op, add, data*) and update PosMap on both sides.
- Wait for server's response.
- Invoke API.Eviction, OST.EvictRecord.

2) Server-side:
- Upon receiving the request, invoke API.Handler.

Transformation between ConcurORAM and Ring ORAM (base ORAM):

Input Data: Stash, PosMap, *MaxStashSize* and batch size c.

Switch to ConcurORAM:

1) Client prepares metadata:
   - Encrypt Stash into TempStash.
   - Pad TempStash with dummies to $MaxStashSize + c$.
   - Send TampStash to server.
   - Inherit dummy metadata and RecMap.
   - Load API.Query and API.Eviction of ConcurORAM.
2) Server-side:
   - Receive TampStash from client.
   - Add TampStash to ConcurORAM's state.
   - Load API.Handler of ConcurORAM.

Switch from ConcurORAM:

1) Server-side:
   - Process the cached requests.
   - Send DRLogSet and latest TemStash to client.
   - Load API.Handler of Ring ORAM.
2) Client recovers metadata:
   - Receives DRLogSet and TemStash from server.
   - Recover Stash from DRLogSet, TempStash.
   - Inherit dummy metadata and RecMap.
   - Load API.Query and API.Eviction of Ring ORAM.

Transformation between Path ORAM and Ring ORAM (base ORAM):

Switch to Path ORAM:

1) Client-side:
   - Inherit dummy metadata and RecMap.
   - Load API.Query and API.Eviction of Path ORAM.
2) Server-side:
   - Load API.Handler of Path ORAM.

Switch from Path ORAM: Identical to switch to Path ORAM, except the APIs are Ring ORAM's.

Figure 3: The basic workflow of V-ORAM. [1]Our considered ORAMs are efficient and implementable through the standard encryption tool AES. [2]The detailed workflow of EvictRecord refer to Algorithm 1, and its input parameters are omitted here.

the client calls API.Eviction to evict the ORAM. Lastly, the client invokes the second phase of Evictrecord to reset the blocks that reach their access limit.

4.4 Performance Analysis

EvictRecord and storage. We first outline the performance of EvictRecord and the stash size of V-ORAM. Their concrete proofs can be found in Appendix B.2. Then, we analyze the transformation costs of our considered ORAMs.

Theorem 1 (Costs of EvictRecord). *For an ORAM with N blocks, bucket size of Z, and block size of B.* EvictRecord *incurs an amortized communication and computation cost of $O(ZB)$, storage costs of $N \log Z$ bit.*

Claim 1. *The stash size of V-ORAM is bounded by $O(\log N)$.*

Transformation costs. Table 2 illustrates the concrete transformation costs between the base ORAM and the other two representative ORAMs. During the transformation of ConcurORAM, the costs come from the transition of DRLogSet and TempStash. The former is the size of $2c^2$ and the latter is $MasStashSize + c$. As for Path ORAM, thanks to RecMap, the transformation cost is the cost of HTTP requests.

5 Planner

Alike previous works [34], our planner aims to provide a generic model able to address complex deployment problems. Here we provide a brief workflow of the planner and defer the discussions on its detailed deployment in Appendix A.4.

Our planner leverages the given dataset, client workload sequence, minimum throughput X_{Sys}, and maximum latency L_{Sys}, to estimate the optimal ORAM parameter, server configuration, and ORAM service sequence.

We simplify the assumptions of ORAM as follows. The server configuration and monetary costs include communica-

Table 2: Transformation cost of two representative ORAMs between base ORAM. "Computation" mainly stands for client encryption, *i.e.*, Enc/Dec. [1]This denotes *MaxStashSize* in Ring ORAM and ConcurORAM.

	Switch to base ORAM		Switch from base ORAM	
	Communication	Computation	Communication	Computation
Path ORAM	$O(1)$	$O(1)$	$O(1)$	$O(1)$
ConcurORAM	$B(2c^2+c+MS$[1]$)$	$B(2c^2+c+MS)$	$B(c+MS)$	$B(c+MS)$

Table 3: The detailed cost of our considered ORAMs. path and bucket denote the data capacity of single path and bucket. [1]The computation cost of XOR operation in server. [2]This includes the costs from PD-ORAM and updating other metadata.

	Download (C_{down})	Upload (C_{up})	Computation (C_{cpu})	Round Complexity
Path ORAM	path	path	$\text{Dec}(C_{\text{down}})+\text{Enc}(C_{\text{up}})$	2
Ring ORAM	$B+(\frac{1}{A}+\frac{1}{S})\text{path}$	$(\frac{1}{A}+\frac{1}{S})\text{path}$	$B\log N$[1]$+\text{Dec}(C_{\text{down}})+\text{Enc}(C_{\text{up}})$	$1+\frac{1}{A}$
ConcurORAM	$B(3+2c+(2+\frac{1}{c})(c+MS))+$ $\frac{1}{c}(\text{path}+\text{bucket}\log c+O(\log N)$[2]$)$	$B(2+4c+2MS)+$ $\frac{1}{c}(\text{path}+O(\log N))$	$B\log N+\text{Dec}(C_{\text{down}})+\text{Enc}(C_{\text{up}})$	$\underbrace{(1+\frac{3}{c})}_{\text{sync}}+\underbrace{\frac{1}{c}(1+\frac{2}{c})}_{\text{async}}$

tion (traffic billing or bandwidth billing), computation (number of CPU cores n_{core}), and storage (data volume). We assume the network and hardware latency (L_{net} & L_{IO}) are constant, the processing speed of a single core P_{cpu} is constant and averages out based on the specific workload. The download and upload speeds P_{down} and P_{up} are constant. The client spending is calculated based on the cloud server price [2, 5].

Modeling ORAMs. The latency of client operation mainly consists of communication and computation latency. We denote the data capacity to be processed of specific operations as C, *i.e.*, computation capacity C_{cpu}, download and upload capacity $C_{\text{down}}, C_{\text{up}}$. The operation latency is defined as:

$$L_{\text{op}} = \frac{C_{\text{cpu}}}{P_{\text{cpu}} \cdot n_{\text{core}}} + \frac{C_{\text{down}}}{P_{\text{down}}} + \frac{C_{\text{up}}}{P_{\text{up}}} + L_{\text{net}} + L_{\text{IO}} \tag{1}$$

where op denotes the client operations, either query of certain ORAM or transformation between two ORAMs.

As for throughput, one cannot simply take the inverse of L_{op} due to batch processing. Thus, for a given batch size *BatchSize*, we define X_{op} as:

$$X_{\text{op}} = BatchSize/L_{\text{op}} \tag{2}$$

Note that our modeling does not provide strong precision guarantees as our assumption ignores subtle details (*e.g.*, processes synchronization, long tail latency). Yet combined with our planner, it does help engineers choose better parameters and brings tangible monetary savings (§6.3).

Choosing ORAMs. Combining the model and the performance requirements, the planner chooses the ORAM sequence O with the optimal monetary cost. We provide detailed access costs of three ORAMs in Table 3.

Parameter selection. We concentrate on the selection of (Z, B), while other parameters follow the security constraints of ORAMs. We assume (Z, B) are selected based on the client datasets before deployment and stay constant throughout V-ORAM. The reasons are: (1) Changing the bucket size needs to adjust the position map and could cause the stash size to explode. (2) The ciphertext is indivisible, altering the block size afterward leads to complex addressing.

The selection comes from that, in practice, the actual file could have different sizes F, *e.g.*, MB-level image files and kB-level text files. Different file sizes require opposite parameter settings. Taking the commonly used factor bandwidth blowup (the ratio of transmitted data to requested block size) [26, 36, 59] of Path ORAM as an example, consider a dataset with total volume of M, when $F < B$, the blowup is $ZB/F \cdot \log(M/ZB)$ decreasing with B; while $F > B$, the blowup is $Z \cdot \log(M/ZB)$ increasing with B. Therefore, there exists an optimal parameter settings according to the data distribution.

Switching ORAMs. The client can estimate the optimal parameter settings of its applied ORAMs. Moreover, combined with Table 2&3, the client can determine whether the current ORAM service meets the performance requirements and decide whether to switch to another ORAM. Planner can be extended to complex scenarios, *e.g.*, given upgrade budgets, outputting resource configuration (bandwidth and CPU core) with maximum performance gains; and adjusting the ORAM sequence if workload changes. Further discussion can be found in Appendix A.4.

6 Evaluation

6.1 Implementation

We implemented a prototype of V-ORAM in Python. We use pycryptodemo v3.20[2] as the block encryption. PRF is realized by sha256 of Python build-in library hashlib.

We also develop prototypes of Path ORAM, Ring ORAM, and ConcurORAM in-house to ensure a unified interface

[2]https://github.com/Legrandin/pycryptodome.

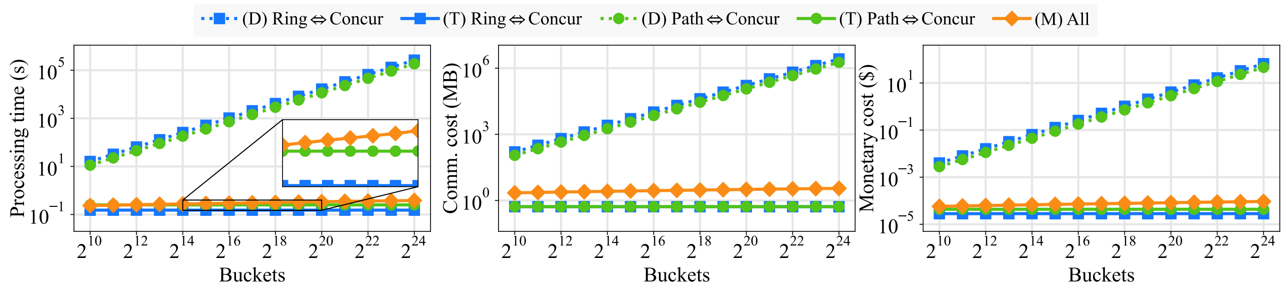

Figure 4: Comparing the performance of OST protocol with two baselines under different ORAM sizes. “D” denotes directly downloading the ORAMs. “M” denotes concurrently maintaining multiple ORAMs. “T” denotes OST protocol.

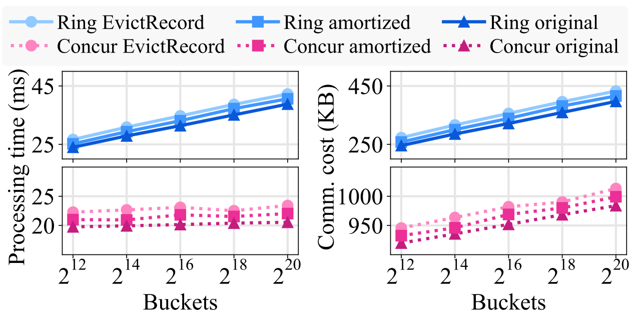

Figure 5: The performance effect of EvictRecord on ORAM services. “Amortized” is the average costs with EvictRecord. “EvictRecord” is the costs of access triggers EvictRecord.

for V-ORAM. Unless otherwise specified, the parameters of ORAMs are set as follows:

- Path ORAM. We set the parameters to meet the security requirements of Path ORAM, *i.e.*, $Z = 8$, B =4kB. It is worth noting that the maximum allowed logical blocks is not the maximum capacity of the tree. The stash size is bounded iff these two values meet specific requirements [64]. In our implementation, for a L-height (from 0 to L) tree and N data blocks, the stash size is well-bounded when $N \leq Z \cdot 2^{L-1}$.
- Ring ORAM. We set the parameters to (Z,A,S)=$(8,8,12)$. In Ring ORAM, the stash size is bounded iff $N \leq A \cdot 2^{L-1}$ [58]. This coincides with the parameters in Path ORAM and applies to our transformation. This is just a set of example parameters, as long as N is the minimum value meeting all requirements, the stash size can be bounded.
- ConcurORAM. We strive to implement a simplified version of ConcurORAM upon the above Ring ORAM. We let the requests come in order and handle them through a single thread. The batch size is set to $c = 8$, aligning with Ring ORAM. As for PosMap, we opt for a smaller Path ORAM, which also maintains the obliviousness in our settings.

Experiment setup. We evaluate the performance of V-ORAM in-promise on a server equipped with a 16-core Intel Xeon Gold 6226R CPU. We set download and upload bandwidth as 10 Mbps and round-trip time is set to immediate.

Compared baselines. We compare OST with two aforementioned baselines. (1) Directly downloads and rebuilds the ORAMs, denoted as “D”. (2) Maintaining multiple ORAM instances, denoted as “M”. While V-ORAM together with OST protocol is denoted as “T”. Note that baseline-M is equivalent to accessing all instances in each query, as the client must generate and upload the re-shuffle blocks.

Measurement. Subsequent experiments involve several indicators, *i.e.*, communication costs, processing time, throughput, and monetary costs. We further explain them as follows:

- Communication costs come from the amount of transmitted data recorded during V-ORAM.
- Processing time (or latency) includes communication and computation time. We measure the processing time by transmitting and processing the recorded data amount for communication and computation.
- Throughput is estimated by dividing the batch size by the processing time of single access. We take the average of 100 accesses as the final throughput.
- Unless otherwise specified, we use traffic billing (0.01 \$/GB) for communication to calculate monetary costs. For ORAMs with server computation, we also add server rental fees (0.544 \$/hr for c6g.4xlarge with 16vCPUs, 32GiB).[3]

6.2 Costs of Service Transformation

Transformation costs. Figure 4 shows the transformation costs under different ORAM sizes. We omit the transformation of Ring ⇔ Path as it only contains API replacements. With more buckets, the costs of baseline-D increase linearly, baseline-M increase logarithmically, while OST is constant.

Regarding the transformation costs of Ring ⇔ Concur. For 1GB logical data (*i.e.*, $N = 2^{16}$), baseline-D costs 1029.6s, 10.0GB, and 0.26\$ to rebuild the ORAMs. While V-ORAM only costs 0.15s, 536kB, and 2.81×10^{-5}\$, saving $10^{3.76}\times$ in processing time, $10^{4.12}\times$ in communication cost, and $10^{3.96}\times$ in monetary costs. We stress that though baseline-M performs

[3]AWS pricing calculator: https://calculator.aws/#/createCalculator/ec2-enhancement

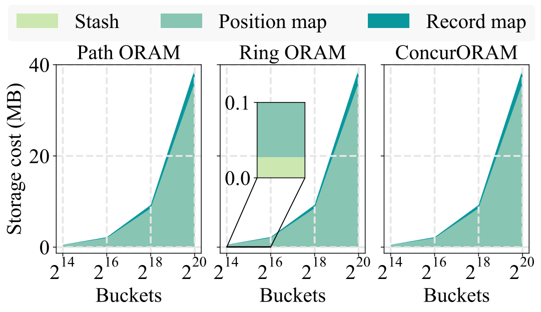

Figure 6: Breakdown storage cost under different ORAMs of each ORAMs during V-ORAM.

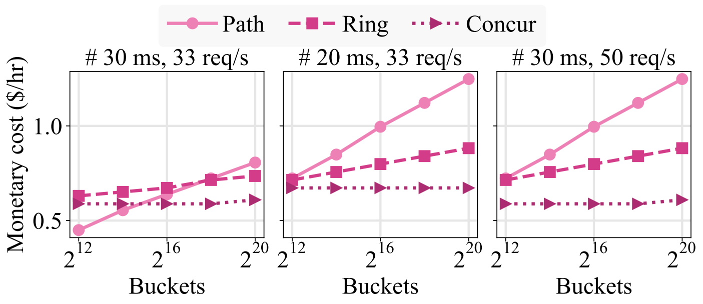

Figure 7: Comparing monetary costs of three ORAMs under different system performance requirements. The communication is billed by bandwidth.

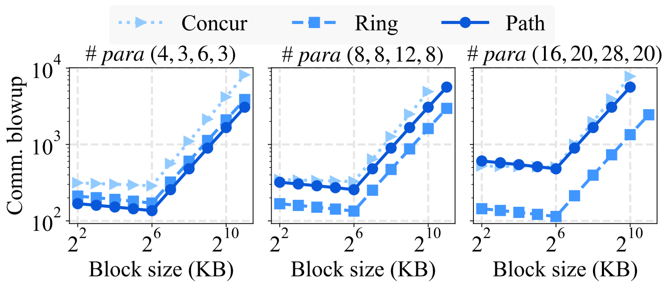

Figure 8: Comparing communication blowup of three ORAMs under different parameter settings. The file size is set to 64kB. "# *para*" denotes the parameters of (Z, A, S, c).

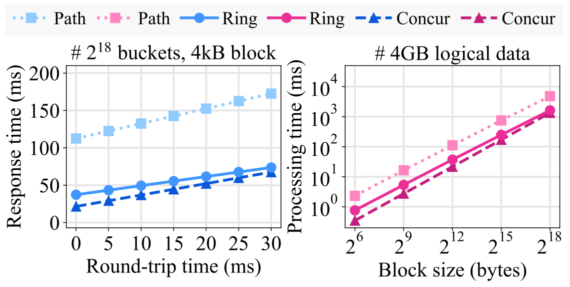

Figure 9: The performance of V-ORAM under realistic settings. We evaluate an V-ORAM with 4GB logical data under various RTT (**Left**) and block size (**Right**).

closely to OST, it is still logarithmic and generates such costs for every access, around 5.36× of Ring ORAM access. It also costs multiplied storage, around 3× of V-ORAM.

The transformation costs of Path ⇔ Concur are equal to the sum of the other two costs. Thus in transformation costs, Path ⇔ Concur perform closely to Ring ⇔ Concur.

EvictRecord costs. Figure 5 shows the costs of EvictRecord in Ring ORAM and ConcurORAM. We omit Path ORAM here as it does not execute our evictions (recall §4.2.1).

To better quantify the costs, we define three costs namely original, EvictRecord, and amortized costs. Recall that OST is invoked together with ORAM API (Figure 3). Thus, compared with the insecure baseline, OST introduces additional costs when EvictRecord occurs. We denote the costs/performance of API without OST as "original", denote the costs/performance when OST triggers eviction as "EvictRecord", and denote actual costs with OST as "amortized".

For Ring ORAM, we measure the average cost for every A access, which includes exactly one API.Eviction. Similarly, ConcurORAM is calculated the every c access. Apparently, EvictRecord costs exceed amortized costs, which in turn surpass original costs. In average, compared with "original", both "EvictRecord" and "amortized" incur extra latency < 5ms and communication < 50kB. Such costs do not scale with ORAM size and are only determined by ORAM parameters.

Storage costs. Figure 6 shows the breakdown storage cost of client-side metadata. Our RecMap can be attached to PosMap with around $N \log Z$ bits storage, *e.g.*, for an ORAM with 1GB logical data ($N = 2^{16}$), RecMap generates storage of < 0.19MB. Here, the stash size is almost constant as our parameter setting ensures low overflow probability (*e.g.*, $\approx 2^{-80}$ for Ring ORAM [58]). While ConcurORAM is measured in every batch, where Stash is already uploaded on the server.

6.3 Planner

We evaluate the planner by choosing cost-effective ORAM based on performance requirements and selecting ORAM parameters with lower communication blowup.

Choosing cost-effective ORAM. Though cloud platforms like AWS and Azure offer instances with high bandwidth, the actual network performance remains constrained by the client-side bandwidth. Consequently, traffic billing does not accurately reflect the relationship between monetary costs and the true performance of ORAM. Thus, we turn to bandwidth billing to estimate the fine-grained evaluation of V-ORAM.

We follow the pricing of AliCloud[4], bandwidth is tiered-pricing, bandwidth $\leq$ 5Mbps is 0.0059\$/Mbps/hr, and band-

[4]Alibaba Cloud, Elastic Compute Service pricing: https://www.alibabacloud.com/en/product/ecs?_p_lc=1

Table 4: Basic statistics of the dataset and the communication blowup for the strategy-recommended block size. [1]File sizes are measured from tripped traces used in evaluations. [2]Total data size here is average cluster size, calculated from [79].

	# Files (k)	Total Data Size (GB)	Average File Size[1] (kB)	75th Percentile File Size (kB)	Estimated Tree Height	Recommended Block Size	Comm. Blowup (k, in Ring ORAM)
MSRC	–	6446.1	26.7	64.0	27	16kB	143.8
AliCloud	–	750.0	29.5	16.0	23	16kB	111.6
Twitter	–	267.71[2]	0.38	0.64	29	256B	169.5
ChestX-ray8	112	42.0	392.4	421.1	16	256kB	95.3
COVIDx	85	29.0	358.8	450.5	15	256kB	98.2

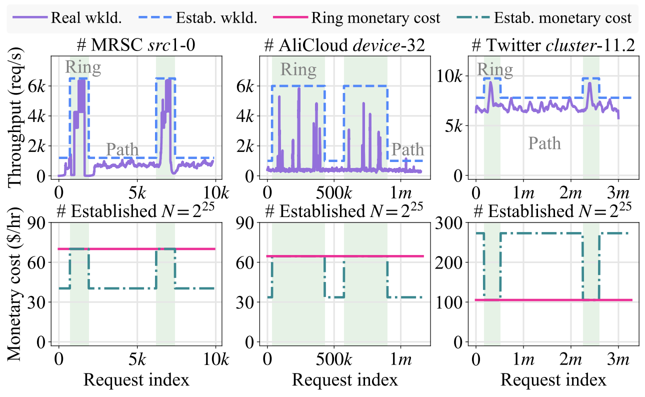

Figure 10: The monetary cost of real-world workloads of MSRC, AliCloud, and Twitter. We estimate N by the max offset of the trace. The communication is billed by bandwidth.

width $>$ 5Mpbs is 0.021\$/Mbps/hr. We also set the server pricing to ecs.u1-c1m2.4xlarge in Virginia, with 16vCPUs, 32GiB. In addition, as the ORAM we selected does not rely on heavy cryptographic tools, computation (≈50Gbps encryption speed) is not a performance bottleneck. The monetary cost is mainly reflected in the different network bandwidths.

Figure 7 shows the estimated monetary cost of ORAM for three sets of system performance requirements. (**Left**) We first set a matched throughput and latency (*i.e.*, $30 \times 33 \approx 1000$). (**Middle**) When the system requires lower latency, Ring ORAM and ConcurORAM are more cost-effective than Path ORAM. ConcurORAM is more cost-effective than Ring ORAM due to its better parallelization. (**Right**) When the system requires higher throughput, CurconrORAM is slightly more cost-effective than middle due to its batch requests.

Selecting ORAM parameters. As established in §5, our focus is on the parameter selection of Z and B, while adhering to the security requirements of the respective ORAMs. In Figure 8, we consider a dataset of size 1GB, and its average file size F is set to 64kB. We assess three different parameter configurations specified in Ring ORAM [58], let B range from 4kB to 1MB, and estimate the communication blowup of three ORAMs. (**Left**) Refer to Table 3, when the values of (A,S,c) are smaller, the coefficient (*i.e.*, $\frac{1}{A}$, $\frac{1}{S}$ and $\frac{1}{c}$) of expensive parts (*i.e.*, path and bucket) is larger in both Ring ORAM and ConcurORAM. Consequently, Path ORAM exhibits a lower blowup under such settings. (**Middle** and **Right**) As (A,S,c) increases, the blowup of Path ORAM also increases due to the growing data volume within path. For Ring ORAM, the effect of the coefficient is more pronounced, leading to decreased blowup. Conversely, ConcurORAM is impacted more by the increase in data volume, resulting in a higher blowup.

As a short summary, the blowup is minimized when $B = F$. Ring ORAM tends to favor larger Z, whereas Path ORAM and ConcurORAM prefer smaller Z. Additionally, Z has a greater impact on Path ORAM compared to ConcurORAM.

6.4 Performance of V-ORAM

To better demonstrate the performance of V-ORAM, we evaluate V-ORAM under the following two realistic settings. As shown in Figure 9, the evaluated ORAMs have logical data of 4GB, equivalent to 2^{18} buckets at 4kB block size.

Realistic latency. (**Left**) We evaluate the response time of V-ORAM under different RTT settings. The amortized round complexities of Path ORAM, Ring ORAM, and ConcurORAM as ORAM services in V-ORAM are around 2, 1.19, and 1.53. Ring ORAM's complexity is slightly higher than its original complexity of 1.13, calculated from Table 3. While ConcurORAM has the same complexity compared with the original, as EvictRecord can be embedded into its evictions [26].

Various block sizes. (**Right**) We evaluate the processing time of V-ORAM under various block sizes. All ORAM services generate less processing time under smaller block sizes. ConcurORAM gradually involves more processing time than Ring ORAM under larger block sizes, aligning with Table 3.

6.5 Real-World Case Study

To better assess the adaptability of V-ORAM, we estimate the monetary savings brought by V-ORAM under real-world workloads. We also simulate two workloads with dynamic throughput and dynamic latency in Appendix A.3.

Real-world workloads. We selected three real-world datasets to further evaluate V-ORAM's application in practice. An overview of these datasets is shown in Table 4. Microsoft Research Cambridge (MSRC) [51] records block-level I/O trace for 36 volumes over 179 disks in 13 servers for 7 days. Alibaba cloud trace (AliCloud) [1] is collected from 1000 disks in its cluster in Beijing over a one-month period in January 2020. Twitter workload [79] is collecting from 153 in-memory cache clusters at Twitter, sifting through over 80TB of data. ChestX-ray8 [74] includes 108,948 chest X-ray images from 32,717 patients at the NIH Clinical Center. COVIDx [71] is the largest chest X-ray dataset containing 13,975 images from 13,870 COVID-19-positive patients. Note that the last two datasets are preprocessed for ML training, making the number of files we summarized higher than the official count.

Parameter selection. As shown in Table 4, we estimate the recommended block sizes based on the file sizes of five datasets and provide the blowup under the standard Ring ORAM settings (see §6.1). The tree heights are estimated by the maximum offset for trace datasets, or the actual dataset size for medical datasets. Note that Twitter does not contain offset and we use average cluster size instead.

When choosing the block size, we compare the two nearest block sizes. For example, the average file size in the MSRC is 26.7kB, so we compare the blowup with block sizes of 16kB and 32kB, resulting in 16kB generating a lower blowup. We observe that MSRC and AliCloud prefer middle block sizes like 16kB as they are collected from disks, Twitter benefits from a smaller block size as it is sourced from cluster cache, and the last two datasets contain numerous image files and are more suitable for larger block sizes.

Monetary cost. As shown in Figure 10, we select the subset of three datasets to evaluate V-ORAM under real-world workload. The subsets are stripped from *src*1-0 in MSRC, *device*-32 in AliCloud, and *cluster*-11.2 from Twitter, spanning 2.06hr, 0.81hr, and 0.12hr, respectively.

(**Up**) We divided the trace into two types of emulated workloads in V-ORAM with different throughputs, marked with dashed lines. We use Ring ORAM for high-throughput workloads and Path ORAM for low-throughput. (**Down**) To estimate the monetary savings of V-ORAM, we compare its monetary costs (dashed lines) with the costs of maintaining Ring ORAM solely (red line). For the former two cases, V-ORAM saves monetary costs of 33.1% and 24.5% respectively. Yet for Twitter, Path ORAM conversely requires more monetary costs. This is because the throughput gap between the two workloads on Twitter is not as significant as in the former two datasets. In that case, our planner assists clients and decides whether to perform transformations.

7 Related Works

Driving ORAM forward. Many studies focus on improving ORAM performance or extending its application scenarios. Regarding performance, numerous research utilizes server computation to reduce communication costs [30, 36, 58] or distribute ORAM across servers/instances to mitigate bottlenecks [34, 62, 63]. Some approaches also leverage trusted hardware to reduce ORAM access latency [16, 34, 50]. As for applications, a series of studies extend ORAM to broader use cases, *e.g.*, searchable encryption [19, 50, 77], secure computation [37, 72, 73], and file-sharing systems [16, 31]. Recently, Chang *et al.* [27] evaluate the performance of various existing ORAMs upon large databases. Yet each of those designs focuses on a specific type of workload and cannot serve the need when the application workload is changing dynamically.

Secure computation frameworks for dynamic workloads. Adapting to dynamic workloads has become a trend for privacy-preserving systems to better align with practice. For example, hybrid multi-party computation (MPC) protocols [22, 29, 38, 45] compile multiple MPC primitives into mixed circuits to support complex programs. Encrypted databases use planners [48, 70] to optimize query costs for dynamic workloads with distinct query types. Meanwhile, Sang and Luo *et al.* [60] design a zero-knowledge proof (ZKP) compiler framework Lian able to compile multiple ZKP protocols into parallelizable chunks. While V-ORAM initially introduces dynamic workloads into ORAM-based databases and provides a planner aiding clients in monetary saving.

8 Conclusion

In this paper, we propose V-ORAM, a versatile and adaptive framework serving dynamic workloads via service transformation. V-ORAM contributes a transformation protocol leveraging Ring ORAM as an intermedia and achieving constant transformation costs regardless of database size. V-ORAM maintains the original security of ORAM as well as the process of transformations. Finally, we validate the practicality of V-ORAM through microbenchmarks and real-world datasets.

Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their valuable comments. This work was supported by the National Key R&D Program of China (No. 2021YFB2900100), the National Science Fund for Distinguished Young Scholars (No. 62025205), and the National Natural Science Foundation of China (No. U22B2022, 62372383).

Ethics Considerations

This paper does not involve human subjects, personal data, or any sensitive information. The three datasets used in this paper are all open-accessible and do not include any personal identities. *Hence, there are no ethical concerns in this paper.*

Open Science

Our source code is available at `https://doi.org/10.5281/zenodo.14732806`, and we also maintain a GitHub repository at `https://github.com/BoZhangCS/V-ORAM`.

References

[1] Alibaba Block Traces. https://github.com/alibaba/block-traces. [Accessed 15-12-2024].

[2] Alibaba Cloud: Cloud Computing Services. https://www.alibabacloud.com/. [Accessed 01-09-2024].

[3] CBI Health & Adastra. https://aws.amazon.com/partners/success/cbi-health-adastra/. [Accessed 01-09-2024].

[4] Cloud-based EHR: Benefits, Migration Tips and Use Cases. https://www.itransition.com/healthcare/ehr/cloud. [Accessed 01-09-2024].

[5] Cloud Computing Services | Amazon Web Services (AWS). https://aws.amazon.com/. [Accessed 01-09-2024].

[6] Global Burden of Disease (GBD). https://www.healthdata.org/research-analysis/gbd. [Accessed 01-09-2024].

[7] Health Catalyst | Healthcare Data and Analytics Technology and Services. https://www.healthcatalyst.com/. [Accessed 01-09-2024].

[8] Health Information Exchange. https://www.healthit.gov/topic/health-it-and-health-information-exchange-basics/health-information-exchange. [Accessed 01-09-2024].

[9] Michigan Health Information Network Shared Services. https://aws.amazon.com/partners/success/mihin-cloudticity/. [Accessed 01-09-2024].

[10] Oracle Health - Reimagine the future of health. https://www.oracle.com/health/. [Accessed 01-09-2024].

[11] Signal-Blog-Technology Deep Dive: Building a Faster ORAM Layer for Enclaves. https://signal.org/blog/building-faster-oram/. [Accessed 01-09-2024].

[12] Technology Deep Dive: Building a Faster ORAM Layer for Enclaves - Signal. https://signal.org/blog/building-faster-oram/. [Accessed 01-09-2024].

[13] Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/. [Accessed 15-12-2024].

[14] The East of England Stroke Telemedicine Partnership & Visionable. https://aws.amazon.com/partners/success/east-england-stroke-telemedicine-partnership-visionable/. [Accessed 01-09-2024].

[15] UtahFS: Encrypted File Storage. https://blog.cloudflare.com/utahfs/. [Accessed 01-09-2024].

[16] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Proc. of NDSS*, 2018.

[17] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. In *Proc. of Springer EUROCRYPT*, 2020.

[18] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. FutORAMa: A Concretely Efficient Hierarchical Oblivious RAM. In *Proc. of ACM CCS*, 2023.

[19] Léonard Assouline and Brice Minaud. Weighted Oblivious RAM, with Applications to Searchable Symmetric Encryption. In *Proc. of Springer EUROCRYPT*, 2023.

[20] Amir Bahmani, Arash Alavi, Thore Buergel, Sushil Upadhyayula, Qiwen Wang, Srinath Krishna Ananthakrishnan, Amir Alavi, Diego Celis, Dan Gillespie, Gregory Young, et al. A Scalable, Secure, and Interoperable Platform for Deep Data-driven Health Management. *Nature Communications*, 2021.

[21] Kornelia Batko and Andrzej Ślęzak. The use of Big Data Analytics in Healthcare. *Journal of Big Data*, 2022.

[22] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hHybrid Protocols for Practical Secure Computation. In *Proc. of ACM CCS*, 2018.

[23] Dingyuan Cao, Mingzhe Zhang, Hang Lu, Xiaochun Ye, Dongrui Fan, Yuezhi Che, and Rujia Wang. Streamline Ring ORAM Accesses through Spatial and Temporal Optimization. In *Proc. of IEEE HPCA*, 2021.

[24] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proc. of ACM CCS*, 2015.

[25] Anrin Chakraborti, Adam J Aviv, Seung Geol Choi, Travis Mayberry, Daniel S Roche, and Radu Sion. rORAM: Efficient Range ORAM with $O(\log^2 N)$ Locality. In *Proc. of NDSS*, 2019.

[26] Anrin Chakraborti and Radu Sion. ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM. In *Proc. of NDSS*, 2019.

[27] Zhao Chang, Dong Xie, and Feifei Li. Oblivious RAM: A Dissection and Experimental Evaluation. In *Proc. of VLDB*, 2016.

[28] Yuezhi Che and Rujia Wang. Multi-Range Supported Oblivious RAM for Efficient Block Data Retrieval. In *Proc. of IEEE HPCA*, 2020.

[29] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S Wahby, Fraser Brown, and Wenting Zheng. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. In *Proc. of IEEE S&P*, 2023.

[30] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE. In *Proc. of ACM CCS*, 2019.

[31] Weikeng Chen and Raluca Ada Popa. Metal: A Metadata-Hiding File-Sharing System. In *Proc. of NDSS*, 2020.

[32] Wen Cheng, Dazou Sang, Lingfang Zeng, Yang Wang, and André Brinkmann. Tianji: Securing a Practical Asynchronous Multi-User ORAM. *IEEE TDSC*, 2023.

[33] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proc. of USENIX OSDI*, 2018.

[34] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *Proc. of ACM SOSP*, 2021.

[35] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An Encrypted Search System with Distributed Trust. In *Proc. of USENIX OSDI*, 2020.

[36] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *Proc. of Springer TCC*, 2016.

[37] Jack Doerner and Abhi Shelat. Scaling ORAM for Secure Computation. In *Proc. of ACM CCS*, 2017.

[38] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. CostCO: An Automatic Cost Modeling Framework for Secure Multi-Party Computation. In *Proc. of IEEE EuroS&P*, 2022.

[39] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proc. of ACM ASPLOS*, 2015.

[40] David Froelicher, Juan R Troncoso-Pastoriza, Jean Louis Raisaro, Michel A Cuendet, Joao Sa Sousa, Hyunghoon Cho, Bonnie Berger, Jacques Fellay, and Jean-Pierre Hubaux. Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption. *Nature Communications*, 2021.

[41] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In *Proc. of Springer CRYPTO*, 2016.

[42] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 1996.

[43] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S^{3}ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing. In *Proc. of ACM CCS*, 2017.

[44] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust. In *Proc. of USENIX NSDI*, 2020.

[45] Muhammad Ishaq, Ana L Milanova, and Vassilis Zikas. Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing. In *Proc. of ACM CCS*, 2019.

[46] Ayya Keshet, Lee Reicher, Noam Bar, and Eran Segal. Wearable and Digital Devices to Monitor and Treat Metabolic Diseases. *Nature Metabolism*, 2023.

[47] Jinxi Kuang, Minghua Shen, Yutong Lu, and Nong Xiao. Exploiting Data Locality in Memory for ORAM to Reduce Memory Access Overheads. In *Proc. of DAC*, 2022.

[48] Elizabeth Margolin, Karan Newatia, Tao Luo, Edo Roth, and Andreas Haeberlen. Arboretum: A Planner for Large-Scale Federated Analytics with Differential Privacy. In *Proc. of ACM SOSP*, 2023.

[49] Surya Mathialagan and Neekon Vafa. MacORAMa: Optimal Oblivious RAM with Integrity. In *Proc. of Springer CRYPTO*, 2023.

[50] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An Efficient Oblivious Search Index. In *Proc. of IEEE S&P*, 2018.

[51] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. of USENIX FAST*, 2008.

[52] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proc. of ACM CCS*, 2015.

[53] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *Proc. of USENIX OSDI*, 2016.

[54] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. of ACM SOSP*, 2011.

[55] Mehrnoosh Raoufi, Jun Yang, Xulong Tang, and Youtao Zhang. AB-ORAM: Constructing adjustable Buckets for Space Reduction in Ring ORAM. In *Proc. of IEEE HPCA*, 2023.

[56] Mehrnoosh Raoufi, Jun Yang, Xulong Tang, and Youtao Zhang. EP-ORAM: Efficient NVM-Friendly Path Eviction for Ring ORAM in Hybrid Memory. In *Proc. of DAC*, 2023.

[57] Mehrnoosh Raoufi, Youtao Zhang, and Jun Yang. IR-ORAM: Path Access Type based Memory Intensity Reduction for Path-ORAM. In *Proc. of IEEE HPCA*, 2022.

[58] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *Proc. of USENIX Security*, 2015.

[59] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming Asynchronicity in Oblivious Data Storage. In *Proc. of IEEE S&P*, 2016.

[60] Yuyang Sang, Ning Luo, Samuel Judson, Ben Chaimberg, Timos Antonopoulos, Xiao Wang, Ruzica Piskac, and Zhong Shao. Ou: Automating the Parallelization of Zero-Knowledge Protocols. In *Proc. of ACM CCS*, 2023.

[61] Elaine Shi, T H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *Proc. of Springer ASIACRYPT*, 2011.

[62] Emil Stefanov and Elaine Shi. Oblivistore: High Performance Oblivious Cloud Storage. In *Proc. of IEEE S&P*, 2013.

[63] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards Practical Oblivious RAM. In *Proc. of NDSS*, 2012.

[64] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proc. of ACM CCS*, 2013.

[65] Reed T Sutton, David Pincock, Daniel C Baumgart, Daniel C Sadowski, Richard N Fedorak, and Karen I Kroeker. An Overview of Clinical Decision Support Systems: Benefits, Risks, and Strategies for Success. *NPJ Digital Medicine*, 2020.

[66] Paul C Tang, Joan S Ash, David W Bates, J Marc Overhage, and Daniel Z Sands. Personal Health Records: Definitions, Benefits, and Strategies for Overcoming Barriers to Adoption. *Journal of the American Medical Informatics Association (JAMIA)*, 2006.

[67] Afonso Tinoco, Sixiang Gao, and Elaine Shi. EnigMap: External-Memory Oblivious Map for Secure Enclaves. In *Proc. of USENIX Security*, 2023.

[68] Shruti Tople, Yaoqi Jia, and Prateek Saxena. PRO-ORAM: Practical Read-Only Oblivious RAM. In *Proc. of USENIX RAID*, 2019.

[69] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. DUORAM: A Bandwidth-Efficient Distributed ORAM for 2-and 3-Party Computation. In *Proc. of USENIX Security*, 2023.

[70] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure Multi-Party Computation on Big Data. In *Proc. of EuroSys*, 2019.

[71] Linda Wang, Zhong Qiu Lin, and Alexander Wong. Covid-Net: A Tailored Deep Convolutional Neural Network Design for Detection of COVID-19 Cases from Chest X-ray Images. *Scientific reports*, 2020.

[72] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proc. of ACM CCS*, 2015.

[73] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *Proc. of ACM CCS*, 2014.

[74] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. ChestX-ray8: Hospital-scale Chest X-ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases. In *Proc. IEEE CVPR*, 2017.

[75] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A Parallel Oblivious File System. In *Proc. of ACM CCS*, 2012.

[76] Xiaohang Wu, Yuxuan Wu, Zhenjun Tu, Zizheng Cao, Miaohong Xu, Yifan Xiang, Duoru Lin, Ling Jin, Lanqin Zhao, Yingzhe Zhang, et al. Cost-effectiveness and Cost-utility of a Digital Technology-driven Hierarchical

Healthcare Screening Pattern in China. *Nature Communications*, 2024.

[77] Zhiqiang Wu and Rui Li. OBI: A Multi-Path Oblivious RAM for Forward-and-Backward-Secure Searchable Encryption. In *Proc. of NDSS*, 2023.

[78] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. Leakage-Abuse Attacks Against Forward and Backward Private Searchable Symmetric Encryption. In *Proc. of ACM CCS*, 2023.

[79] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[80] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient Random Access in Multi-Party Computation. In *Proc. of IEEE S&P*, 2016.

[81] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *Proc. of USENIX Security*, 2016.

A Extended Content

A.1 Protocols of Involved ORAMs

We now briefly introduce the processes of three ORAMs. These introductions are not exhaustive since we are just providing a basic understanding. For ConcurORAM, we only describe its query process here. QueryLog is used to record the identity of current queries, which is not mentioned in the main text. There are also other complex metadata in CocnurORAM, which once again demonstrates the necessity of our padding request before transformation (§4.2).

Path ORAM. As shown in Algorithm 2, the query includes:

- Line 1−2: Get the path x of target block from PosMap and remap the block to a random path.
- Line 3: Read all buckets on path x and add them into Stash (denoted as S here).
- Line 4−6: Get the target block from Stash. Update the data if the operation is write.
- Line 7−11: ***Instant Eviction.*** Traverse the buckets in the path, randomly select and write back the blocks in Stash that can be placed into the current bucket.
- Line 12: Return the data.

Ring ORAM. As shown in Algorithm 3, the query includes:

- Line 1−4: Get the path x of target block from PosMap and remap the block to a random path.
- Line 5−11: Read the XORed block on path x. The block is either in path or Stash. Then return the data if the operation is read, or update the data if the operation is write. Lastly, add the block into Stash.
- Line 12−13: ***Periodic Eviction.*** Increase the counter round and perform the eviction if round $\bmod A = 0$.
- Line 14: EarlyReshuffle reshuffles the buckets that exhaust the dummies.

ConcurORAM. As shown in Algorithm 4, the query includes:

- Line 1−2: Update the QueryLog in a blocking manner and get the current query identifier i.
- Line 3−6: Obliviously read the blocks in StashSet, path, and DRLogSet.
- Line 7−9: Return the data if the operation is read, update the data if the operation is write, and update DRLogSet.
- Line 10−13: In the last request of the batch, ConcurORAM reshuffles the metadata and prepares for future requests.

Algorithm 2: Path ORAM (Figure 1 in [64]).

```
   // Access(add, op, data')
1  $x \leftarrow$ PosMap[add];
2  PosMap[add] $\leftarrow$ UniformRandom$(0, 2^L - 1)$;
3  $S \leftarrow S \cup$ ReadPath$(x)$
4  data $\leftarrow$ Read block add from $S$;
5  for op = write do
6      $S \leftarrow (S - \{(\text{add}, \text{data})\}) \cup \{(\text{add}, \text{data}')\}$;
7  for $l \in \{L, L-1, \ldots, 0\}$ do
8      $S \leftarrow$ {Blocks that can be placed in $P(x,l)$};
9      $S \leftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$;
10     $S \leftarrow S - S'$;
11     WriteBucket$(P(x,l), S')$;
12 return data;
```

A.2 Description of Workloads

- General storage: Many products or scenarios leverage long-term storage, *e.g.*, collecting global healthcare data [6], data outsourcing service [7, 10] Its primary requirement is monetary costs, with no specific performance requirement. This workload is treated as compatible with object storage (*e.g.*, AWS S3) since its friendly maintenance fees, in which the server is a pure storage provider.
- Real-time updates: Numerous emergencies require immediate responses, *e.g.*, clinical decision-making [65], vital signs monitoring, and heart attack surveillance [21]. In these scenarios, sensors monitor the patient and transmit the record to the healthcare facility. Therefore, the key requirement of this workload is to minimize the latency.
- Parallel access: Scenarios such as health information exchange [8, 66] or federated analysis [40, 48] involve multiple clients sharing a single ORAM across healthcare institutions. Traditional ORAM schemes enable parallel access

Algorithm 3: Ring ORAM (Algorithm 1 in [58]).

```
   // Access(add, op, data')
1  Global/persistent variables: round;
2  $x' \leftarrow$ UniformRandom$(0, 2^L - 1)$;
3  $x \leftarrow$ PosMap[add];
4  PosMap[add] $\leftarrow x'$;
5  data $\leftarrow$ ReadPath$(x,$ add$)$;
6  if data $= \perp$ then
7      // If block add is not found on path x,
           it must be in Stash.
8      data $\leftarrow$ Read and remove add from Stash;
9  if op $=$ read then return data to client;
10 if op $=$ write then data $\leftarrow$ data$'$;
11 Stash $\leftarrow$ Stash $\cup$ (add, $x'$, data);
12 round $\leftarrow$ round $+ 1 \mod A$;
13 if round $= 0$ then EvictPath();
14 EarlyReshuffle$(l)$;
```

Algorithm 4: ConcurORAM (Algorithm 7 in [26]).

```
   // Query(id)
1  Update query log QueryLog (with mutex);
2  $i \leftarrow len($QueryLog$) \mod c$; // Query identifier
3  Stash $\leftarrow$ Stash $\cup$ ReadStashSet(id, $i$);
4  $x \leftarrow$ PosMap[id];
5  path $\leftarrow$ ReadPath$(x)$;
6  blk_DRL $\leftarrow$ ReadDRLogSet(id);
7  Target block $blk$ must in blk_DRL $\cup$ path $\cup$ Stash;
8  Update $blk$ for write;
9  WriteDRLogSet$(blk, i)$;
10 if $i = c - 1$ then
11     Update and reshuffle DRLogSet;
12     Update and reshuffle Stashset;
13     Initialize new query log for future queries;
```

by relying on a trusted proxy or client interactions. In contrast, ConcurORAM removes these dependencies by using server-side metadata and mutexes to achieve parallelism.

- Resource constrained: On wearable or mobile devices with limited storage, it may not be feasible to store entire metadata, *e.g.*, PosMap. Correspondingly, ORAMs with recursion could reduce the client storage costs to $O(\log N)$.
- Specialized functionality: The above workloads vary on the performance and resource requirements, while the corresponding ORAMs also only support read and write operations. In addition, other workloads could require specialized functions supported by general databases, *e.g.*, keyword searching in encrypted EHRs [41, 50, 77], or directly calculating on ciphertext [69, 72, 73].

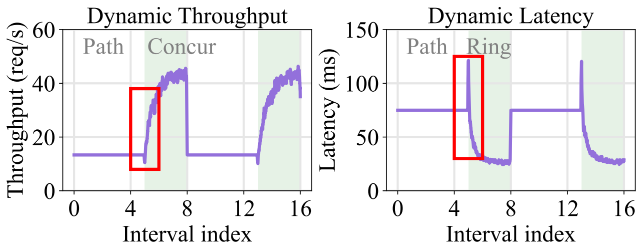

Figure 11: Simulated workloads for dynamic throughput and latency. Each interval is set to 2^{11} requests. Red-boxed areas denote the temporary performance effect of EvictRecord. After that, the performance is restored to its original level.

A.3 Simulated Workloads

In Figure 11, we simulated two dynamic workloads: dynamic throughput and dynamic latency. We used ConcurORAM for high throughput, Ring ORAM for real-time updates, and Path ORAM for general storage. V-ORAM effectively adapts to both dynamic workloads, adjusting the ORAM service efficiently. During the period marked by the red box, EvictRecord is shuffling buckets from the previous ORAMs, leading to a temporary performance drop. Switching to Path ORAM does not cause a performance drop because EvictRecord does not evict buckets in Path ORAM.

A.4 Discussion

Generalizations of V-ORAM. The primary goal and contribution of V-ORAM is to address dynamic workloads in practice by initially offering an efficient and secure solution. We are aware of other efficient structures, including hierarchical ORAM [17, 18] and square-root ORAM [80], catering to specialized workloads. For example, MacORAMa [49] is a hierarchical ORAM designed for malicious scenarios, where the server exhibits malicious behavior or data errors and requires integrity verification. PRO-ORAM [68] is a square-root ORAM designed for read-only scenarios, protecting solely the obliviousness of read operations.

We acknowledge that V-ORAM currently is unable to transform between these structures. Tackling these transformations introduces unexplored challenges such as switching between different encrypted data structures and eviction/access protocols, and remains a future research direction.

Unpredictable workloads. The workloads in this paper are assumed predictable, *i.e.*, the clients know the workloads prior. It is natural to extend V-ORAM for unpredictable workloads. For example, in workloads with dynamic throughput, we can set a threshold, say twice the low-peak throughput. Once V-ORAM detects that a certain amount of requests consistently exceed the threshold, it switches to an ORAM with higher throughput and vice versa.

Concurrent workloads. Although our transformation costs are independent of database size, V-ORAM still causes service interruptions. Exploring how to support concurrent workloads is an interesting task. Combining Ring ORAM and Path ORAM could be a promising direction, given their $O(1)$ transformation costs in both theoretical and empirical.

Different ciphertext constructions. When switching between ORAMs with different ciphertext types, data blocks need to be re-encrypted into the required format. We can let the server perform the reconstruction. In initialization, the client generates and sends the server master key mk_S to the server. During the transformation, the server directly re-encrypts the previous ciphertexts via mk_S. Once the retrieving the block, the client strips encryption layers to obtain the plaintext. Though reconstruction increases the client's waiting time, it does not increase the communication costs. This is also secure as we do not protect the service type.

Applications of planner. In the main text, we assume the client knows the data distribution and workload in advance and selects suitable ORAM parameters and service sequences before deployment. The planner can also address more complex deployment scenarios. (1) If the current ORAM fails to meet performance requirements, the planner enables the client to upgrade server resources within a set budget according to Table 3, *e.g.*, acquiring higher bandwidth or additional CPU cores. (2) If the workload changes unexpectedly, *e.g.*, with bursty requests, the client can estimate the new performance requirements and use the planner to select the optimal ORAM service. The client can adopt the former strategy if the selected ORAM still falls short.

B Theoretical Analysis

B.1 Security Analysis

B.1.1 Security of EvictRecord

Theorem 2. *Given a client access sequence $\vec{y}$, and a random sequence $\vec{z}$ with the same length and service types, the* EvictRecord *algorithm of V-ORAM described in §4 is secure according to Definition 1.*

Proof. For an access sequence $\vec{y}$ of length m, within the non-EvictRecord portion, our invariant suggests that the probability of an access pattern leak is $\Pr'(\vec{y}) = negl(m)$.

In the EvictRecord phase, regarding data confidentiality, if the encryption scheme employed is CCA-2 secure [5], then the blocks written are indistinguishable from new random blocks to the server, thereby not revealing the content of data.

As for obliviousness, EvictRecord merges the read blocks into Stash and randomly shuffles them before writing back. The server is unable to track specific blocks confidently. Therefore, the server can only randomly guess the access pattern. For k EvictRecords, the probability that the server correctly infers the access pattern is $\Pr(\vec{y}) = (2^l)^{-k} \cdot \Pr'(\vec{y})$. Since $k \propto m$, the overall probability is $negl(m)$. □

[5]We assume Adv has unlimited access to AES, as normally the employed algorithm is publicly available. Thus, AES is required to be CCA-2 secure.

B.1.2 Security of V-ORAM

Theorem 3. *Given a pseudorandom function, a CCA-2 secure encryption scheme, and the oblivious* EvictRecord *satisfying Theorem 2, V-ORAM framework described in §4 is secure according to Definition 2.*

Proof. V-ORAM includes three main protocols (Init, Transform, Access), where Init initializes the storage system, Transform switches the current ORAM service, and Access queries and evicts the data. In the real experiment, we let Π execute the above protocols, while in the ideal experiment, we design simulators SimInit, SimTransform, and SimAccess. Figure 12 describes the algorithms of the above protocols. We further explain the syntax and design details of the protocols:

Protocols in **Real** experiments:

- $\underline{\hat{D} \leftarrow \Pi.\mathsf{Init}(mk, D, \mathsf{Encoder}, \mathsf{API})}$: This protocol encrypts the Adv-generated dataset D to $\hat{D}$ according to the initialization algorithm, and initialize Stash and PosMap.
- $\underline{\tau_T \leftarrow \Pi.\mathsf{Transform}(\mathsf{sid})}$: This protocol performs transformations in Figure 3. PrepMeta and RetrMeta represent the metadata preparation and recovery in ConcurORAM. τ_{T_0} represents the data written to the server when preparing metadata, and τ_{T_1} represents the read data when recovering metadata. Finally, this protocol returns the data set τ_T visible to Adv.
- $\underline{\tau_A \leftarrow \Pi.\mathsf{Access}(\mathsf{op}, \mathsf{add}, \mathsf{data})}$: The client executes the APIs of the current ORAM service to access the data with address add. τ_{A_0} represents the data read by the client in the query, τ_{A_1} and τ_{A_2} represent the data read or written back in API.Eviction and EvictRecord, respectively. Finally, this protocol returns the data set τ_A visible to Adv.

Protocols in **Ideal** experiments:

- $\underline{\hat{D} \leftarrow \mathsf{SimInit}(\mathsf{Encoder}, \mathsf{API})}$: Simulates the initialization of the system, where the used dataset here is made up of random data blocks with the same format and length.
- $\underline{\tau_T \leftarrow \mathsf{SimTransform}(\mathsf{sid})}$: Simulates the process of client transformation to sid. SimPrepMeta includes the encryption of metadata in ConcurORAM, which has random content with the same structure and length. PosMap is stored in PD-ORAM as well yet the content is randomly generated. SimRetrMeta reads and decrypts the server-side metadata. Since the decryption takes place locally and is invisible to Adv, SimRetrMeta merely simulates read operation and ignores the retrieved content. Finally, SimTransform returns the data set τ_T visible to Adv.
- $\underline{\tau_A \leftarrow \mathsf{SimAccess}()}$: Simulates the procedures of client access. SimQuery is identical to API.Query and is secure according to Definition 1 as we assumed in our threat model (§3.1). SimEviction and SimEvictRecord are identical to

```
$\hat{D} \leftarrow \Pi.\mathsf{Init}(mk, D, \mathsf{Encoder}, \mathsf{API})$:
    for $i \leftarrow 0$ to $N-1$ do
        $\hat{D}[i] = \mathsf{Encoder}_B.\mathsf{Enc}(D[i], \mathsf{sid}_b)$;
    Stash $\leftarrow \{\}$ and initialize PosMap;
    return $\hat{D}$, PosMap;

[1]$\tau_T \leftarrow \Pi.\mathsf{Transform}(\mathsf{sid})$:
    // Metadata preparation
    $\tau_{T_0} \leftarrow \mathsf{PrepMeta}()$;        // Switch to
    WaitSync();                                          // Switch back
    $\tau_{T_1} \leftarrow \mathsf{RetrMeta}()$;
    // API replacement
    Server loads API.Handler of sid;
    Client loads API.Query, API.Eviction of sid;
    return $\tau_T \leftarrow \tau_{T_0} \cup \tau_{T_1}$;

$\tau_A \leftarrow \Pi.\mathsf{Access}(\mathsf{op}, \mathsf{a}, \mathsf{data})$:
    Client access;
    $\tau_{A_0} \leftarrow$ Buckets read in API.Query;
    $\tau_{A_1} \leftarrow$ Buckets written in API.Eviction;
    $\tau_{A_2} \leftarrow$ Buckets written in EvictRecord;
    return $\tau_A \leftarrow \tau_{A_0} \cup \tau_{A_1} \cup \tau_{A_2}$;
```

```
$\hat{D} \leftarrow \mathsf{SimInit}(\mathsf{Encoder}, \mathsf{API})$:
    for $i \leftarrow 0$ to $N-1$ do
        $\hat{D}[i] \leftarrow \$\ \mathsf{sid}_b \,\|\, \{0,1\}^B$;
    Stash $\leftarrow \{\}$ and randomly initialize PosMap;
    return $\hat{D}$, PosMap;

$\tau_T \leftarrow \mathsf{SimTransform}(\mathsf{sid})$:
    // Metadata preparation
    $\tau_{T_0} \leftarrow \mathsf{SimPrepMeta}()$;     // Switch to
    WaitSync();                                          // Switch back
    $\tau_{T_1} \leftarrow \mathsf{SimRetrMeta}()$;
    // API replacement
    Server loads API.Handler of sid;
    Client loads API.Query, API.Eviction of sid;
    return $\tau_T \leftarrow \tau_{T_0} \cup \tau_{T_1}$;

$\tau_A \leftarrow \mathsf{SimAccess}()$:
    $\tau_{A_0} \leftarrow$ SimQuery;
    $\tau_{A_1} \leftarrow$ SimEviction;
    $\tau_{A_2} \leftarrow$ SimEvictRecord;
    return $\tau_A \leftarrow \tau_{A_0} \cup \tau_{A_1} \cup \tau_{A_2}$;
```

Figure 12: Simulator algorithms of real and ideal experiments. [1]τ denotes the requested (being read or written) blocks.

real experiments, except the data written back is random. Finally, SimAccess returns the data set τ_A visible to Adv.

Figure 13 describes the flow of the real and ideal experiments. Both experiments first take security parameter and public information in V-ORAM as input, *i.e.*, λ and (Encoder, API). In line with other ORAM designs [18, 34, 77], we let Adv generate data sets D and $\vec{y}$. During the simulations, we maintain *service* to denote the current ORAM service. At the end of the experiments, Adv gives its judgment *bit* based on the generated $\hat{D}$, τ and public information (Encoder, API).

According to Definition 2, our V-ORAM framework is secure as long as the advantage of Adv to successfully distinguish real and ideal experiments is $negl(\lambda)$.

Security. In the initialization phase, since the employed AES algorithm is CCA-2 secure, the advantage of Adv in distinguishing between an encrypted $\hat{D}$ and a random dataset with the same length is $negl(\lambda)$.

In terms of service, both experiments maintain the same service and transform services according to $\vec{y}$ provided by Adv. Therefore, as long as the adversary cannot distinguish the transformation with a non-negligible advantage, it also cannot distinguish the experiments based on the service status.

The transformation can not be distinguished by Adv with a non-negligible advantage as: (1) For EvictRecord, the simulator could execute SimEvictRecord at the appropriate time according to the access limit of each service (considered as public information of sid). Moreover, since the written blocks are re-encrypted and randomly shuffled, the advantage for Adv to distinguish EvictRecord and SimEvictRecord is $negl(\lambda)$. (2) For transformation of ConcurORAM, the client downloads/uploads the metadata at each transformation, which is identical regardless of simulator inputs. Moreover, Adv cannot distinguish the uploaded metadata between the random data generated from simulations with non-negligible advantage. Besides, due to the security of PD-ORAM [75], Adv can neither learn the content of PosMap nor distinguish the two experiments based on the access sequence to PosMap. Thus, the advantage for Adv to distinguish the real and ideal transformation of ConcurORAM is $negl(\lambda)$. (3) For transformation of Path ORAM, our protocol does not prepare any metadata visible to Adv and directly switches the APIs of V-ORAM. Thus, Adv can not distinguish the real and ideal transformation of Path ORAM.

In summary, the advantage of Adv distinguishing the real from the ideal experiments at any stage of our system is a negligible probability $negl(\lambda)$. According to Definition 1&2, our V-ORAM framework is secure. □

B.2 Performance Analysis

B.2.1 Proof of Theorem 1

Proof. ***Communication cost***: Consider a simulated V-ORAM′ only equipped with EvictRecord, *i.e.*, the blocks would not evicted by other eviction. This is an unsecured variant only for analysis. Apparently, in V-ORAM′, the access count of blocks would not be reset by eviction of original ORAMs. Thereby the inferred communication and computa-

```
$bit \leftarrow \textbf{Real}^{\Pi}_{\mathsf{Adv}}(\lambda, \mathsf{Encoder}, \mathsf{API})$:
    // Initialization
    $D, \vec{y} \leftarrow \$\ \mathsf{Adv}$;
    $mk \leftarrow \$\ \{0,1\}^{\lambda}$;
    $\hat{D}, \mathsf{PosMap} \leftarrow \Pi.\mathsf{Init}(mk, D, \mathsf{Encoder}, \mathsf{API})$;
    Initialize PosMap and its APIs;
    // Access
    $service \leftarrow \mathsf{sid}_b$, $\tau \leftarrow \{\}$;
    for $i \leftarrow 0$ to $|\vec{y}|$ do
        $(\mathsf{op}_i, \mathsf{add}_i, \mathsf{data}_i, \mathsf{sid}_i) \leftarrow \vec{y}[i]$;
        if $service \neq \mathsf{sid}_i$ then
            $\tau_T \leftarrow \Pi.\mathsf{Transform}(\mathsf{sid}_i)$;
            $service \leftarrow \mathsf{sid}_i$;
        $\tau_A \leftarrow \Pi.\mathsf{Access}(\mathsf{op}_i, \mathsf{add}_i, \mathsf{data}_i)$;
        $\tau[i] \leftarrow (\tau_T, \tau_A)$;
    return $bit \leftarrow \mathsf{Adv}(\hat{D}, \tau, \mathsf{Encoder}, \mathsf{API})$;
```

```
$bit \leftarrow \textbf{Ideal}^{\mathsf{Sim}}_{\mathsf{Adv}}(\lambda, \mathsf{Encoder}, \mathsf{API})$:
    // Initialization
    $D, \vec{y} \leftarrow \$\ \mathsf{Adv}$;
    $mk \leftarrow \$\ \{0,1\}^{\lambda}$;
    $\vec{z} \leftarrow$ Random sequence with same length and sid of $\vec{y}$;
    $\hat{D}, \mathsf{PosMap} \leftarrow \mathsf{SimInit}(\mathsf{Encoder}, \mathsf{API})$;
    Initialize PosMap and its APIs;
    // Access
    $service \leftarrow \mathsf{sid}_b$, $\tau \leftarrow \{\}$;
    for $i \leftarrow 0$ to $|\vec{z}|$ do
        $(\mathsf{op}_i, \mathsf{add}_i, \mathsf{data}_i, \mathsf{sid}_i) \leftarrow \vec{z}[i]$;
        if $service \neq \mathsf{sid}_i$ then
            $\tau_T \leftarrow \mathsf{SimTransform}(\mathsf{sid}_i)$;
            $service \leftarrow \mathsf{sid}_i$;
        $\tau_A \leftarrow \mathsf{SimAccess}()$;
        $\tau[i] \leftarrow (\tau_T, \tau_A)$;
    return $bit \leftarrow \mathsf{Adv}(\hat{D}, \tau, \mathsf{Encoder}, \mathsf{API})$;
```

Figure 13: Real and ideal experiments of V-ORAM framework.

tion cost of EvictRecord is *greater* than the actual costs.

We consider the randomized uniform accesses of each block. Thus the probability of buckets at layer i (counted from root node 0 to leaf node $l = \log N - 1$) being accessed is 2^{-i}. For k accesses, the evicted times of buckets at layer i is $\frac{k}{lim} \cdot 2^{-i}$. Normally, the query reads a certain path from a leaf node to a root node, leading to the overall evicted buckets of $\frac{k}{lim} \cdot (2^0 + 2^{-1} + \cdots + 2^{-\log N+1}) = \frac{2k}{lim} \cdot (1 - \frac{1}{N})$. Therefor, the amortized communication and computation cost of EvictRecord is less than $\frac{2ZB}{lim} \cdot (1 - \frac{1}{N})$, *i.e.*, $O(ZB)$.

Storage cost: The record map can be attached after the entries of the position map. Each entry would cost at most $\log lim$ bits. In Ring ORAM and other ORAMs with delayed eviction [30, 36], lim has the same magnitude of Z, leading to the overall storage costs of $N \log Z$ bit. □

B.2.2 Proof of Claim 1

Proof. We formulate it as a "Claim" rather than a "Theorem", as it is tricky to formally bound the stash size. As well as the multiple services we converted also complicate the proof. However, thanks to our cleverly designed base ORAM mechanism, the stash size of V-ORAM equipped with the services considered in this paper can be derived from existing proofs.

Firstly, for ConcurORAM that built upon Ring ORAM, the proof of ConcurORAM is essentially the proof of Ring ORAM with EvictRecord. Note that EvictRecord is equivalent to resetting the access count of buckets in advance, *i.e.*, a smaller number of dummy blocks $S' \leq S$.

Ring ORAM formally proves that [58], given stash size R, the eviction constant A, and the number of cached blocks st, the probability of Ring ORAM exceeding the stash is:

$$\Pr[\mathsf{st} > R] < \frac{(a/Z)^R}{1 - e^{-q}}$$

where $a = A/2 < Z$ and $q = Z \ln(Z/a) + a - Z - \ln 4 > 0$. This shows that the stash size is only related to (Z, A). The selection of S does not affect the stash size!

Moreover, the general storage Path ORAM is equivalent to $S' = 0$, which also guarantees the above theorem. In summary, the stash size of V-ORAM is bounded by $O(\log N)$. □