MVP-ORAM: a Wait-free Concurrent ORAM for Confidential BFT Storage

Robin Vassantlal Hasan Heydari Bernardo Ferreira Alysson Bessani
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

***Abstract*—It is well known that encryption alone is not enough to protect data privacy. Access patterns, revealed when operations are performed, can also be leveraged in inference attacks. Oblivious RAM (ORAM) hides access patterns by making client requests oblivious. However, existing protocols are still limited in supporting concurrent clients and Byzantine fault tolerance (BFT). We present MVP-ORAM, the *first wait-free ORAM protocol* that supports concurrent fail-prone clients. In contrast to previous works, MVP-ORAM avoids using trusted proxies, which necessitate additional security assumptions, and concurrency control mechanisms based on inter-client communication or distributed locks, which limit overall throughput and the capability to tolerate faulty clients. Instead, MVP-ORAM enables clients to perform concurrent requests and merge conflicting updates as they happen, satisfying wait-freedom, i.e., clients make progress *independently of the performance or failures of other clients*. Since wait and collision freedom are fundamentally contradictory goals that cannot be achieved simultaneously in an asynchronous concurrent ORAM service, we define a weaker notion of obliviousness that depends on the application workload and number of concurrent clients, and prove MVP-ORAM is *secure in practical scenarios where clients perform skewed block accesses*. By being wait-free, MVP-ORAM can be seamlessly integrated into existing confidential BFT data stores, creating the first BFT ORAM construction. We implement MVP-ORAM on top of a confidential BFT data store and show *our prototype can process hundreds of 4KB accesses per second* in modern clouds.**

I. Introduction

Context and motivation. Byzantine Fault-Tolerant State Machine Replication (BFT SMR) is a classical technique to implement fault- and intrusion-tolerant replicated services with strong consistency [1], [2]. The technique attracted significant attention in the last decade due to the emergence of decentralized systems and blockchains [3], [4], which can be seen as replicated state machines. BFT SMR systems offer *data integrity* and *availability* guarantees, even if up to t of the n replicas are compromised. Some works have additionally studied how to make BFT SMR systems offer *confidentiality* [5]–[9], g uaranteeing *d ata s ecrecy* e ven in the presence of Byzantine faults. This is typically achieved by combining symmetric encryption with secret sharing [10], [11], a technique where a secret (e.g., an encryption key) is split into n shares (one for each server) and any subset of $t+1$ of them is required for recovering it.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241809
www.ndss-symposium.org

However, encryption, even if augmented with secret sharing, is not enough to ensure data secrecy. Data access patterns, revealed when clients perform operations, can also be leveraged in inference attacks, sometimes with disastrous consequences [12]–[15]. Access patterns typically leaked in a storage service include which entries are accessed, when, how often, if they are accessed with other entries, and whether they are being read or written.

State of the art. Oblivious RAM (ORAM) [16]–[18] is a cryptographic technique whose objective is to conceal access patterns and make them *oblivious*, i.e., indistinguishable from each other. However, it was initially designed for a single CPU accessing its RAM (or a client accessing its server). Recent works have studied its suitability for supporting multiple CPUs/clients in different yet related lines of work, known as parallel ORAM [19]–[24] and multi-client ORAM [25]–[29]. However, these works typically require inter-client communication for synchronization [24], distributed locks [29], or trusted proxies/hardware [26], all of which severely limit concurrency or require additional security assumptions. Concurrently, researchers have also studied how multiple servers can be leveraged in ORAM [30]–[34] to reduce client-server bandwidth. To the best of our knowledge, QuORAM [35] is the only multi-server protocol that increases ORAM availability but only addresses server crashes. Hence, to date, no ORAM protocol can conceal access patterns in Byzantine-resilient systems, and existing protocols cannot be extended to meet the requirements of BFT SMR without sacrificing client fault tolerance.

Problem statement. We address the problem of designing *practical BFT-replicated storage services that can hide data access patterns*, going beyond existing works on confidential BFT systems (e.g., [5]–[9]). We aim to design a replicated ORAM service in which replicas can be subject to Byzantine failures while concurrently accessed by fail-prone clients. In more detail, we are interested in a setting where concurrent clients access a replicated storage service in which (1) servers are subject to Byzantine faults, (2) multiple clients can concurrently access shared data stored in servers without external coordination or waiting for each other, and (3) data access patterns remain oblivious. In this setting, a particularly important property to achieve is *wait-freedom* [36], which states that every client operation (ORAM access) is guaranteed to finish in a finite number of steps. We stress the practical relevance of this property, as wait-free services are more robust than lock-based services [37] (the norm in the concurrent ORAM

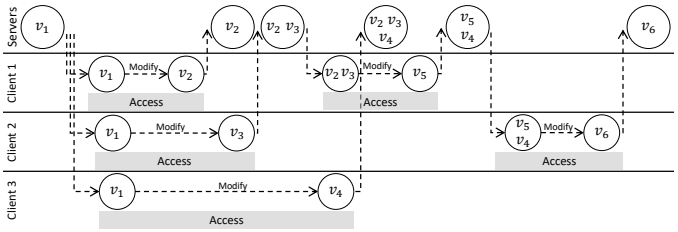

Fig. 1. Multiple versions being created by concurrent accesses and later merged in MVP-ORAM.

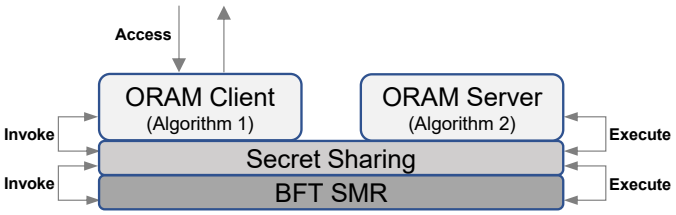

Fig. 2. MVP-ORAM protocol stack.

literature), as clients can finish their accesses independently of the delays and faults of other clients.

Our solution. This paper presents Multi-Version Path ORAM (MVP-ORAM), the first ORAM protocol designed explicitly for concealing access patterns in BFT SMR systems while achieving wait-freedom. MVP-ORAM is based on Path ORAM [38], a simple ORAM protocol where a single client accesses a single server to store encrypted data blocks organized as a binary tree. We selected Path ORAM as a starting point due to its low number of client-server round-trips per access, compared to more recent solutions (e.g., [39], [40]), which is crucial in BFT SMR systems. In Path ORAM, the client accesses data by reading and writing a whole path of the tree where the block of interest is located. To access the correct path, the client maintains a table mapping block addresses to paths in the tree where the blocks are stored. The client also maintains an expected small stash of blocks that have temporarily overflowed from the tree. Path ORAM matches the required bandwidth lower bound of $O(\log N)$ for obliviously accessing a data store with N blocks [41].

MVP-ORAM improves Path ORAM in two fundamental ways. First and foremost, it supports multiple clients concurrently accessing data while satisfying strong consistency [42]. Contrary to previous works on parallel ORAM, which use locks and other inter-client coordination mechanisms to ensure consistency and security, we aim to support client-independent, *wait-free* ORAM accesses, as required in SMR-based services [2]. To the best of our knowledge, no ORAM satisfies this property.

To support wait-freedom, MVP-ORAM encrypts and stores the position map and stash in the server, along with the tree. Clients read the position map to define the block's access path and then request the path and stash from the server. With the ORAM data structures stored on the server, MVP-ORAM supports *multiple clients concurrently accessing data by managing multiple versions of each data structure*. More specifically, servers start with a single version of each ORAM data structure, but concurrent clients can read and generate new versions of these data structures, which are later merged by clients during their access, as illustrated in Fig. 1. Notice that, although our goal is to support BFT ORAM, MVP-ORAM's wait-free design is of independent interest, as no existing single-server scheme has its unique set of features.

Second, MVP-ORAM relaxes the trust assumption on the servers by tolerating Byzantine failures. More specifically, besides allowing servers to observe their internal state, as in the semi-honest model used in Path ORAM and most ORAM schemes, MVP-ORAM tolerates t malicious servers by employing a BFT SMR protocol to replicate a deterministic ORAM service in $n > 3t$ servers.

To make our oblivious BFT data store feature-complete, we tackle the problem of managing the shared keys used to encrypt server data. Contrary to previous works on the multi-client setting, which assume that encryption keys are shared between clients in some way, MVP-ORAM integrates a secret-sharing framework to distribute encryption keys through the servers alongside the ORAM state [7], [9]. Fig. 2 illustrates our construction, where MVP-ORAM runs on top of secret sharing and BFT SMR.

Security and performance. From a security perspective, achieving wait-freedom in ORAM introduces new challenges, as without client synchronization, it becomes impossible to ensure that no two clients access the same block at the same time, a property known in ORAM as *collision-freedom* [19]. Indeed, we argue that in asynchronous networks it is fundamentally impossible to conciliate wait- and collision-freedom. To circumvent this limitation and increase security, when a client wants to access a block, MVP-ORAM chooses and requests at random any of the paths that contain it. Additionally, to increase the number of paths available to access a block, evictions move the most popular blocks closer to the tree root. Assuming ORAM accesses follow a Zipfian distribution, meaning that a very high percentage of accesses are done to a very small percentage of blocks (as has been shown to happen in many natural and digital systems [43]–[45]), this solution allows MVP-ORAM to preserve wait-freedom while its security approximates that of collision-free ORAMs. Nonetheless, collisions can still happen (e.g., two clients simultaneously access a leaf block), even if with low probability. We address this limitation by proposing a variant of MVP-ORAM, which uses dummy requests to ensure obliviouness at the cost of performance and synchrony assumptions.

In terms of performance, MVP-ORAM requires an amount of bandwidth linearly proportional to the number of servers and quadratic in the number of *active concurrent clients* accessing the system at a time, being thus an *adaptive wait-free construction* [46].

We implemented MVP-ORAM on top of COBRA [9], an open-source confidential BFT SMR library, and evaluated its performance. Our results show that our prototype can process

more than 350 (resp. 700) accesses per second with a latency of less than 140 ms (resp. 70 ms) in a system with 10 servers (resp. a single server) and 50 concurrent clients. This shows MVP-ORAM can achieve performance numbers in line with other practical (concurrent/fault-tolerant) protocols [29], [35]. Our implementation is open source [47], and our results are fully reproducible, as described in Appendix B.

Contributions. We claim the following contributions:

1) We initiate the study of the problem of implementing a wait-free ORAM (§III);
2) We design MVP-ORAM, the first asynchronous wait-free ORAM supporting concurrent fail-prone clients (§V). Besides detailing the basic protocol that satisfies a weaker version of obliviousness, we present a stronger version of MVP-ORAM that provides standard parallel ORAM security (§VIII);
3) We use MVP-ORAM to hide access patterns of a confidential BFT SMR-based storage service, providing the first ORAM that tolerates Byzantine-faulty servers (§VI);
4) We present a detailed analysis of MVP-ORAM's stash size and bandwidth requirements, along with security proofs that demonstrate both its correctness and obliviousness (§VII);
5) We implement and evaluate MVP-ORAM to show its performance in practical settings (§IX).

II. Background and Related Work

Confidential BFT. The seminal work on intrusion tolerance by Fraga and Powell [48] was the first to consider information scattering for protecting data confidentiality in a replicated synchronous system. Later works like Secure Store [49] and CODEX [50] ensured confidentiality, integrity, and availability of stored data in asynchronous systems by using Byzantine quorum protocols [51] together with secret sharing [10]. To the best of our knowledge, DepSpace [5] was the first work to use secret sharing for achieving confidentiality in a BFT SMR system. Although DepSpace and follow-up works such as Belisarius [6] achieved performance similar to non-confidential BFT SMR, neither of them supported features required in practice, such as replica state recovery, replica group reconfiguration, or protection against a mobile adversary. The same can be said about works adding confidentiality based on secret sharing to blockchains (e.g., [8]). Basu et al. [7] partially solved this by introducing a confidential state recovery protocol for static BFT SMR. COBRA [9] proposed the first confidential BFT SMR system with all the practical features required by these. Nevertheless, none of these works tackles the problem of hiding access patterns, as an adversary can still observe which data entries are being accessed. This feature can, for example, be used to improve the privacy of a service like Arke [52], which provides confidential contact discovery using a BFT storage service.

Classical ORAM. Oblivious RAM was first introduced by Goldreich and Ostrovsky [16]–[18], in the context of software protection. Subsequent works improved its efficiency in different scenarios [53]–[56]. In the 2010s, with the rise of cloud computing, renewed interest in ORAM led to new improvements, including Path ORAM [38], which was the first ORAM protocol capable of achieving logarithmic bandwidth overhead (shown to be optimal in the storage-only server setting [18], [41]). Subsequent works reduced this overhead to $O(1)$ by assuming server computations [39], [57]. However, these typically require homomorphic encryption to be secure, thus reducing bandwidth overhead but decreasing overall performance. Other works improved performance by using trusted execution environments [58], [59], but this requires shifting trust from well-established cryptographic assumptions to closed-source solutions from hardware manufacturers. More recently, researchers have revisited the original hierarchical ORAM of Goldreich and Ostrovsky to make it more practical [40], [60], [61]. In all these works, a single client accesses a single server.

Multi-client and Parallel ORAM. Recent works have explored how to support multiple concurrent clients in ORAM in different yet related research lines known as multi-client [25]–[29], [35], [62], [63] and parallel ORAM [19]–[24]. Multi-client ORAM focuses on the client-server model, while parallel ORAM focuses on multi-core CPUs, but both try to solve the same problem: how to support concurrency in ORAM accesses. Here, the challenge is twofold: first, obliviousness should be ensured not only for access sequences from individual clients but also between clients. This means that if all clients decide to access the same data block at the same time, the resulting ORAM accesses should still look random and independent to the server. The second is how to efficiently deal with concurrency and synchronize local ORAM client data without making the resulting system inherently sequential. Many works solved these issues by introducing a trusted, confidential proxy (either in the network or in the server) between the clients and the server [26], [35], [62], [63]. The ORAM protocol is then executed between the trusted component and the server, essentially serializing requests and synchronizing client data. Other works avoided the trusted component by relying on inter-client communication [19]–[23] or lock-based distributed algorithms [28], [29] to serialize conflicting concurrent requests. However, these approaches limit client concurrency and prevent wait-freedom from being achieved. Moreover, most of these works only support a single server, making them vulnerable to server faults.

Multi-server ORAM. Another related research vector is the use of multiple ORAM servers to reduce client bandwidth requirements [30]–[34], [63]. Each server plays a critical role in these works, so fault tolerance is not supported. As far as we know, QuORAM [35] is the only ORAM protocol that uses multiple servers to tolerate faults. However, it only tolerates benign (crash) faults and requires trusted proxies attached to servers, an additional strong security assumption. Indeed, in QuORAM, each server plus proxy constitutes an isolated ORAM instance, and a variant of the classical ABD protocol [64] is used to replicate read/write operations on those proxies, which act as (single) clients to their ORAM servers.

The research gap: BFT ORAM. As evidenced by our previous discussion, no ORAM protocol can be integrated "as is" into a confidential BFT system to hide access patterns. The state-of-the-art in ORAM fault tolerance is QuORAM [35], but it only tolerates crashes and requires trusted execution support on servers, i.e., each server needs a trusted proxy. Extending it to tolerate Byzantine faults seems doable, but it would still require trusted proxies. Regarding multi-client support without using proxies, ConcurORAM [29] is the state-of-the-art. However, it has three main limitations. First, it heavily relies on multi-threading and locks at the server, which introduces nondeterminism, significantly complicating replication (e.g., [65]). Second, wait-freedom is impossible to achieve using locks, seriously compromising client fault tolerance. Third, it requires 18 client-server interactions to complete one ORAM access (query and eviction), which, if replicated, would require Byzantine consensus for totally ordering each request. This large number of client-server iterations is also a limitation of a recent optimal parallel ORAM by Asharov et al. [24]. In contrast, by extending Path ORAM (which only requires two round-trips per access) to keep multiple versions of the ORAM state, MVP-ORAM supports concurrent clients without using locks and requiring just three round-trips per ORAM access (see Fig. 3), making it thus more appropriate to be integrated into confidential BFT SMR systems.

Notice that the need for a multi-client ORAM free of locks or inter-client coordination to replicate using BFT SMR made us address another research gap of independent interest: the lack of wait-free multi-client ORAMs.

III. Model and Definitions

System model. We consider a fully connected distributed system in which processes are divided into two sets: a set of n servers/replicas $\Sigma = \{r_1, r_2, \ldots, r_n\}$, and an unbounded set of clients $\Gamma = \{c_1, c_2, \ldots\}$. We assume a trusted setup in which each replica and client has a unique identifier that can be verified by every other process through standard means, e.g., a public key infrastructure. We also assume the system has sufficient synchrony to implement BFT SMR and consensus. For instance, our prototype requires a *partially synchronous model* [66] in which the system is asynchronous until some *unknown* global stabilization time, after which it becomes synchronous, with known time bounds for computation and communication.[1] Finally, every pair of processes communicates through *private and authenticated fair links*, i.e., messages can be delayed but not forever.

Service model. Clients access the replicated storage service, which contains N data blocks, by sending requests and receiving replies to/from the service replicas. Servers globally store a two-part state $\Omega = \langle C, P \rangle$. The common state C comprises ORAM data, and the private state P comprises encryption keys used to encrypt C. Each server r_i locally maintains a state $\Omega_i = \langle C, P_i \rangle$. The common state C is encrypted and replicated across all servers, i.e., all servers store the same state, while the private state is distributed using the secret sharing protocol. Hence, r_i's private state P_i comprises shares of the encryption keys. The functionality of our ORAM service offers a single operation, described in the following way:

- $\langle data \rangle = \mathsf{access}(c_i, op, addr, data^*)$: client c_i invokes access to read or write block addressed by $addr$, i.e., c_i invokes $\mathsf{access}(c_i, read, addr, \perp)$ to read the block and $\mathsf{access}(c_i, write, addr, data)$ to write $data$ to the block.

Finally, we assume applications using our storage service generate a *skewed block access pattern*. More specifically, we assume that storage clients *collectively* induce an access pattern in which a small fraction of the stored blocks are accessed much more frequently than the others. This skewed pattern, typically modeled by a Zipfian distribution [69], is commonly observed in datasets [43] and in storage systems accesses (e.g., [44], [70], [71]), being modelled in popular storage systems benchmarks [72], [73]. This skewness is exploited in most real systems through caching and load-balancing techniques. In this paper, we use it to characterize the obliviousness of wait-free ORAM.

Adversary model. We consider an adversary that can fully control a fraction of the replicas and the scheduling of messages, but has limited access to clients. In particular, we assume that the adversary can maliciously corrupt some of the replicas and crash clients, but can not inject concurrent queries, as that would allow it to force collisions between client accesses. This assumption is somewhat similar to the models of Pancake [74] and Waffle [75], which consider a *passive persistent adversary* that can observe all accesses but cannot inject its own queries. We believe this model accurately captures the typical security guarantees of BFT data stores, where a set of semi-trusted clients store shared data using untrusted servers. Nonetheless, in the Strong MVP-ORAM variant (§VIII), we remove this assumption and consider that the adversary additionally can inject concurrent queries and force collisions.

More formally, we consider a probabilistic polynomial-time adaptive adversary that can control the network and may at any time decide to corrupt a fraction $t < n/3$ of the replicas or crash clients. Corrupted replicas can deviate arbitrarily from the protocol, i.e., they are prone to Byzantine failures. Such replicas are said to be faulty or corrupted. A process that is not faulty is said to be correct or honest. The adversary can learn about the private state that corrupted replicas store and the access patterns of operations received. Clients are assumed to be honest, so they can only fail by crashing and cannot be influenced by the adversary in any other way.

As in other oblivious datastores and confidential BFT services [7], [9], [74], [75], we do not consider fully malicious clients, as there is little point in protecting the confidentiality of a service if malicious clients have permission to access the data. In practice, our service supports multiple ORAMs, each of which is shared by a set of mutually trusted clients. Nonetheless, this restriction can be alleviated through mechanisms for verifiable computation, such as ZK-Proofs [28], [76]

[1] MVP-ORAM construction is oblivious to the used BFT SMR implementation. Nothing precludes MVP-ORAM from being implemented on top of asynchronous protocols (e.g., [67], [68]).

or MPC-based proxies [77], [78]. We leave the integration of these techniques with MVP-ORAM for future work.

Security definition. Beyond ensuring the Safety, Liveness, and Secrecy properties that are standard in confidential BFT services [5], [7], [9], MVP-ORAM additionally aims at ensuring *Obliviousness* (i.e., Access Pattern Secrecy) [16].

Safety (i.e., Linearizability), requires the replicated service to emulate a centralized service [42]; Liveness (i.e., Wait-Freedom) requires all correct client requests to be executed [36]; and Secrecy (i.e., Confidentiality) requires that no private information about the stored data be leaked as long as the failure threshold of the system is respected [9].

As for Obliviousness, we start with the definition from parallel ORAM [19], [20], [79]: given any two sequences of parallel operations $\overrightarrow{y_1}$ and $\overrightarrow{y_2}$ of equal length, they should look indistinguishable to the adversary, except with negligible probability in N. This definition requires the ORAM to be *collision-free* [19], i.e., no two clients ever access the same address concurrently. However, we argue that in asynchronous networks, no ORAM protocol can simultaneously be collision- and wait-free, as the former is impossible to achieve without client synchronization (e.g., distributed locks [29], inter-client communication [19]), which in turn prevents the latter (since a single client failure can prevent others from progressing).

Hence, we propose a new obliviousness definition for asynchronous wait-free ORAM: the indistinguishability between $\overrightarrow{y_1}$ and $\overrightarrow{y_2}$ is characterized by the statistical distance of their access patterns, which depends not only on the ORAM size N, but also the number of concurrent clients c and the distribution of concurrent accesses $\mathcal{D}$, sampled from the universe of all accesses $\mathcal{U}$, from which both $\overrightarrow{y_1}, \overrightarrow{y_2}$ are themselves sampled.

Since the adversary can control the number of concurrent clients accessing the service through network scheduling, we assume the worst-case scenario in which all c clients are accessing the ORAM simultaneously. This increases the likelihood that multiple clients will request the same block concurrently. We define a *timestep* as the interval from the start of the first concurrent access to the end of the last concurrent access among the group of c clients. With this notion, we now provide the security definition for wait-free ORAM:

Definition 1 (Asynchronous Wait-Free ORAM). *Given $c, N \in \mathbb{N}$ and $\mathcal{D} \in \mathcal{U}$, let $\overrightarrow{b_e} = \{b_i\}_{i \in \{1,\dots,c\}}$ denote a set of c concurrent operations in timestep e and $\overrightarrow{y} = (\overrightarrow{b_1}, \overrightarrow{b_2}, \dots)$ denote a sequence of such concurrent operations in each timestep. Protocol Π is an Asynchronous Wait-Free Oblivious Parallel RAM (or simply Asynchronous Wait-Free ORAM) if there exists a function μ such that:*

- ***Correctness:** Given $\overrightarrow{y} \xleftarrow{\$} \mathcal{D}$, the execution of Π returns the last written version of each block requested in $\overrightarrow{y}$ (i.e., for each block, the version with the highest timestamp).*
- ***Obliviousness:** Let $A(\overrightarrow{y})$ denote the access pattern generated by Π when $\overrightarrow{y}$ is executed. We say Π is secure if for any two sequences of concurrent operations $\overrightarrow{y_1}$, $\overrightarrow{y_2} \xleftarrow{\$} \mathcal{D}$ of the same length, with inputs chosen by clients, the statistical distance between $A(\overrightarrow{y_1})$ and $A(\overrightarrow{y_2})$ is bounded by $\mu(N, c, \mathcal{D})$.*

This definition is weaker than the one used in traditional parallel ORAM (e.g., [19], [20], [79]), as it does not allow the adversary to inject queries and it depends on additional security parameters that may make μ non-negligible, namely c and $\mathcal{D}$. In MVP-ORAM, μ will be negligible if, per timestep, a single client accesses the ORAM or multiple clients access different blocks. However, if multiple clients access the same block concurrently, μ may not be necessarily negligible, although it can be arbitrarily small (see §VII-A).

In applications where this may be a problem (e.g., if $\mathcal{D}$ is expected to be uniform instead of Zipfian) and if network synchrony can be assumed, our Strong MVP-ORAM (§VIII) can be used instead, sacrificing performance and asynchrony but fulfilling wait-freedom and parallel ORAM security.

IV. A First Multi-Client ORAM Protocol

We start by presenting a first attempt to design a multi-client ORAM protocol based on Path ORAM [38] that does not require distributed locks, inter-client communication, or trusted proxies to serialize client requests. This first protocol achieves a liveness property known as *obstruction-freedom* [80], meaning a client can finish an invoked ORAM access only if all other clients stop making new requests.

A. Path ORAM

Path ORAM is a simple protocol in which a client invokes an access operation to read or write data blocks from/to an ORAM server. The server keeps N fixed-size *blocks*, each associated with a logical *address*, in a *binary tree* of height L and 2^L leaves. Each tree node is called a *bucket* and contains Z blocks. The client locally maintains a *position map* associating each block to a path in the tree. Let $l \in \{0, \dots, 2^L - 1\}$ be a leaf node of the binary tree. A path $\mathcal{P}_l = \{\mathcal{B}_0, \dots, \mathcal{B}_L\}$, contains all buckets from the root to node l. The client also maintains a *stash* with blocks waiting to be written to the tree because their paths are full.

To access a block, the client starts by discovering its path in the position map and requests all buckets of the path from the server, adding them to the stash. It then reads/modifies the block, changes its path at random, refreshes the encryption of all fetched blocks, and attempts to evict all blocks in the stash back to the server. Evictions follow a read-path eviction strategy, meaning that blocks can only be written back in the intersection between their assigned paths and the read path.

This simple scheme guarantees obliviousness by reading a whole path per access, instead of a single block, and by randomly changing the path of blocks each time they are accessed. It requires a bandwidth of $O(\log N)$ bits, matching the lower bound for storage-only ORAM protocols [41], and only requires two round-trips per access, the lowest amongst practical ORAMs [39], [40], [61]. This is an important metric in BFT SMR, and the main reason for selecting Path ORAM as a starting point, as server requests must be totally ordered

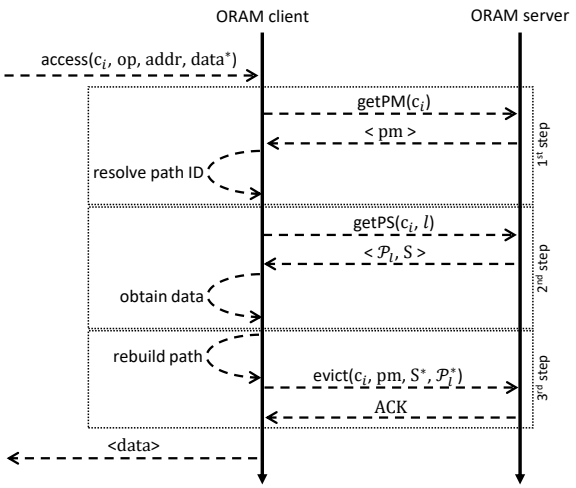

Fig. 3. Simple Multi-Client Path ORAM protocol.

via consensus before being processed by the servers, thereby making each access very costly.

B. Extending Path ORAM to Multiple Clients

Two issues must be addressed to extend Path ORAM to support multiple clients. First, Path ORAM requires the client to keep the position map and stash, so multiple clients must have access to shared, up-to-date versions of these data structures. Second, concurrency must be managed carefully, not only to avoid concurrent accesses leaking information, but also to prevent tree inconsistencies.

At a high level, our first multi-client Path ORAM addresses these challenges by moving client storage (encrypted) to the server and having clients fetch and update it during their access with the help of the server to manage concurrency.

Fig. 3 illustrates our first multi-client Path ORAM. The protocol requires servers to implement the following functionality:

- $pm \leftarrow \mathsf{getPM}(c_i)$: if there is no *active client*, sets client c_i as active (i.e., started an access operation) and retrieves the current position map pm; else, it returns $\perp$.
- $\langle \mathcal{P}_l, S \rangle \leftarrow \mathsf{getPS}(c_i, l)$: if c_i is still the *active client*, the server returns path $\mathcal{P}_l$ and stash S; otherwise, the server returns $\perp$.
- $\mathsf{evict}(c_i, pm, S^*, \mathcal{P}_l^*)$: evicts updated position map pm, new stash S^*, and new path $\mathcal{P}_l^*$ to the server. If c_i is still the active client, the server stores the received data and sets the active client to $\perp$.

Let b be a block with address $addr$ that client c_i wants to access, and pm and S be the position map and stash, respectively. All data received/sent from/to the server must be encrypted/decrypted, but we omit these operations for simplicity. The access operation (§III) has three steps:

1) To access b, c_i must first discover its path. This is done by invoking getPM from the server to retrieve pm and accessing $pm[addr]$ to obtain path id l. If the server returns $\perp$, c_i retries after a random back-off time.
2) Client c_i invokes getPS to obtain path $\mathcal{P}_l$ and stash S from the server. Since c_i is still the active client, the server returns the requested data. Client c_i adds all blocks from $\mathcal{P}_l$ and S to a working set W, reads/writes block b, and assigns a new random path to b in pm.
3) The access ends with c_i evicting the blocks from W. The client first populates a new path $\mathcal{P}_l^*$ with the blocks from W. The path $\mathcal{P}_l^*$ is filled from leaf to root with blocks in the intersection between l and their paths. Overflowing blocks are stored in a new stash S^*. Then c_i sends the new path $\mathcal{P}_l^*$ to the server, along with the updated pm and S^*, by invoking evict. Upon receiving this request, the server checks if c_i is still the active client and, if so, replaces its path $\mathcal{P}_l$ in the tree by $\mathcal{P}_l^*$, its position map by pm, and its stash by S^*.

Although each individual step is *atomic* at the server, the access is not, as it requires three steps, and different clients can interleave these steps, interrupting accesses one from another. As a result, this first protocol only ensures a client completes its access if no other client accesses the ORAM concurrently, satisfying obstruction-freedom [80]. Another consequence of this design is that a failure of a client during an access, which will never end, might block other clients forever.

V. Multi-Version Path ORAM

Extending the previous protocol to support concurrent wait-free accesses requires addressing two fundamental problems. The first is how to avoid breaking obliviousness on concurrent accesses to the same address. Indeed, when clients access the same address in the same timestep, they will request the same path. This breaks *collision-freedom* [19], as it allows the server to infer that clients may be accessing the same address, even if it can not pinpoint exactly which one is being accessed. The second problem is how to preserve data consistency between concurrent evictions. Since access operations are not atomic nor serialized through client synchronization, multiple (possibly conflicting) versions of the tree will be generated.

To tackle these problems, we propose *Multi-Version Path ORAM* (MVP-ORAM). In further detail, to tackle the first problem, we make the server store the exact slot of the bucket where the block is located in the position map, instead of its path. This key idea allows clients to retrieve a block through any path that passes through the slot where the block is stored. Specifically, when clients want to access a block b, they first discover its location sl using the position map. Then, they extend the location to one of the paths that pass through sl, and use that path to retrieve the block. Since clients select these paths randomly, multiple clients accessing the same block will request different paths with increasing probability as the block is higher in the tree.

To further increase the number of available paths, we keep the last accessed block in the stash, meaning it can be accessed again using any path, and evict the most frequently accessed blocks to higher levels of the tree. Specifically, when performing an access, the accessed block always goes to the stash (if it is not already there) along with the non-dummy

blocks from Z slots uniformly selected at random. Then, these Z slots are filled with Z random blocks previously in the stash, i.e., we swap Z blocks between the stash and the accessed path. The constant Z is important to bound the stash size. Additionally, after the swap, we reorder blocks in the path by placing the most frequently accessed blocks higher in the tree. The result is that after each access, the most popular blocks in a skewed access pattern will be accessible through many paths, improving the ORAM obliviousness.

To address the second problem, we enable clients to complete their access in isolation and store updates as new versions of the tree on the server. During an access, each client fetches the existing versions currently stored in the server and merges them into a single, updated tree (as illustrated in Fig. 1). Note that in practice, clients only need to merge paths and stashes that they will retrieve in an access, rather than entire trees.

When multiple versions accessed by clients are merged together, such a merge needs to be done (1) without losing block updates,[2] (2) by keeping more frequently accessed blocks on higher levels of the tree, and (3) by avoiding block duplication in different tree nodes during concurrent evictions. To satisfy these requirements, the server assigns a sequence number to each access during the invocation of its Server.getPM. This sequence number is used to create a logical *block timestamp* $ts_b = \langle v, a, s\rangle$ for each block b touched during an access, where *version* v is the sequence number of the last write on this block, *access* a is the sequence number of the last read or write of this block, and *sequence* s is the sequence number of the last time the block was moved. For instance, if the sequence number of an access to block b is x, ts_b after the operation will be $\langle x, x, x\rangle$ if b is written or $ts_b = \langle \_, x, x\rangle$ if b is read. Further, any other block that changed its slot during this access' eviction will have its timestamp set to $\langle \_, \_, x\rangle$.

Using three values on the block timestamp ensures that the merge requirements 1-3 described above are satisfied. When clients perform an access, they may retrieve multiple paths and stashes with different block versions, and merge them into a single version consistent with the highest timestamp found on the position map for each block, respecting the following rule:

$$\langle v, a, s\rangle > \langle v', a', s'\rangle \implies (v > v') \vee (v = v' \wedge a > a') \vee (v = v' \wedge a = a' \wedge s > s').$$

A. The MVP-ORAM Protocol

Here we present a detailed description of the MVP-ORAM protocol. The data structures used in the protocol are summarized in Table I.

Client c_i accesses block b, identified by address $addr$, by invoking function access, described in Algorithm 1. The function accesses b in three steps, just like the protocol of the previous section, by invoking the server functions specified in Algorithm 2. The local functions invoked by clients to merge concurrent versions and create an updated version of the ORAM state are underlined in Algorithm 1 and illustrated in Fig. 4. Their formal specification is deferred to Appendix A. Note that in the algorithms, all data clients send to servers (except for i and l, which are basic information) is encrypted, but this is omitted for simplicity.

[2]Linerizability [42], or register atomicity, requires a read executed after a write to always return the last update on the stored data.

TABLE I MVP-ORAM DATA STRUCTURES.

Data Structure	Description
Block	A tuple $\langle addr, data, ts\rangle$, where $addr \in \{0, \ldots, N-1\}$ is an address identifying the block, $data$ is the data of the block, and ts is the block timestamp $ts = \langle v, a, s\rangle$.
Slot	Identifier of a position in a binary tree where a real or dummy block is stored.
Bucket	Set of Z slots indexed from 0 to $Z-1$.
Multi-Version Tree	A binary tree of height $L > 0$, where each node contains a set of buckets created concurrently with blocks of different versions. A path $\mathcal{P}_l$ contains the nodes from leaf l to the tree's root. We use the notation $\mathcal{P}_l(sl)$ to denote the set of blocks on slot sl of a path from leaf l.
Position Map	This structure maps block addresses to the current block slot and logical timestamp. $pm[addr] = \langle sl, ts\rangle$ means block with address $addr$ is stored in slot sl with timestamp ts.
Path Map	Set of tuples $M_l = \{\langle addr, sl, ts\rangle, ...\}$ with the position map updates performed during an access, i.e., for each updated block $addr$, its new slot sl and timestamp ts.
Stash	List of overflowing blocks.
ORAM State	A tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{H}_{pathMaps}\rangle$ that stores a multi-version tree $\mathcal{T}$ (with multiple buckets per node), a set of stashes $\mathcal{S}$, one for each version of $\mathcal{T}$, and a set of path maps $\mathcal{H}_{pathMaps} = \{M_l, ...\}$ that when consolidated define a position map pm. We use notation $\mathcal{T}(l)$ and $\mathcal{T}(l, sl)$ to denote path $\mathcal{P}_l$ in $\mathcal{T}$ and slot sl of that path.
Context	A list of ORAM States, one for each concurrent client that started an access and has not yet finished it.

Algorithm 1: MVP-ORAM client c_i.

```
1  Function access(c_i, op, addr, data*)
2      ⟨H_pathMaps, seq⟩ ← Server.getPM(c_i)
3      pm ← consolidatePathMaps(H_pathMaps)
4      ⟨sl, _⟩ ← pm[addr]
5      l ← random path that passes through slot sl
6      ⟨P_l, S⟩ ← Server.getPS(c_i, l)
7      W ← mergePathStashes(P_l, S, pm)
8      if op = write then
9          data ← data*; v ← seq
10     else
11         ⟨_, data, ⟨v, _, _⟩⟩ ← W[addr]
12     W[addr] ← ⟨addr, data, ⟨v, seq, seq⟩⟩
13     ⟨P_l*, S, M_l⟩ ← populatePath(W, l, addr, pm, seq)
14     Server.evict(c_i, M_l, P_l*, S)
15     return data
```

First step (A1, L2-L5). In the first step, client c_i will define a path to retrieve b. It starts by invoking Server.getPM, sending its id and receiving the history of path maps $\mathcal{H}_{pathMaps}$ and sequence number seq that identifies this access. $\mathcal{H}_{pathMaps}$ contains the location updates of blocks evicted so far. In practice, c_i retrieves new updates since its last access.

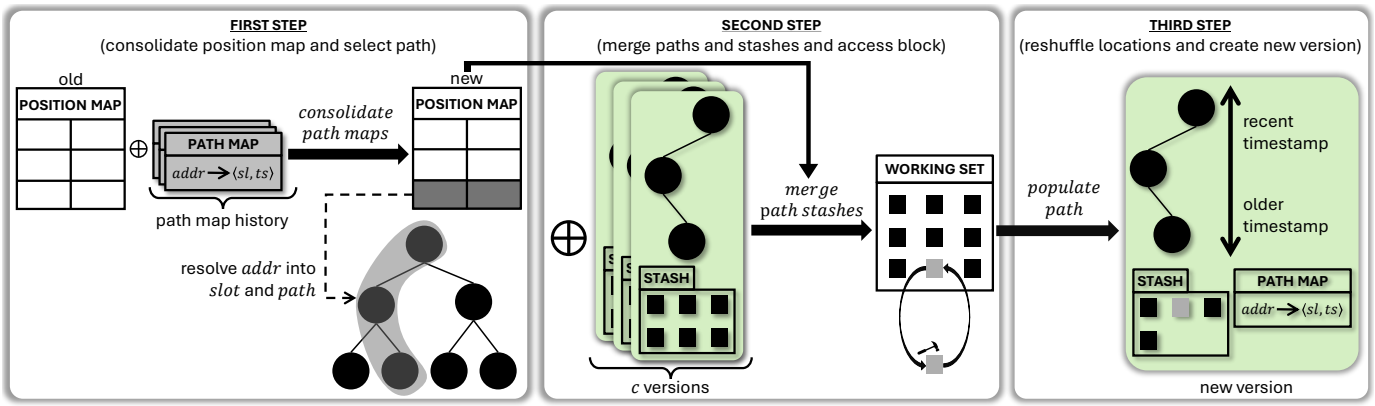

Fig. 4. Overview of the MVP-ORAM protocol.

Algorithm 2: MVP-ORAM server.

```
1  Procedure setup(T, S)
2      oramState ← ⟨T, S, ∅⟩; nextSeq ← 1
3      ∀c_i ∈ Γ : context[c_i] ← ⊥
4  Function getPM(c_i)
5      seq ← nextSeq; nextSeq ← nextSeq + 1
6      context[c_i] ← oramState
7      ⟨_, _, H_pathMaps⟩ ← oramState
8      return ⟨H_pathMaps, seq⟩
9  Function getPS(c_i, l)
10     ⟨T, S, _⟩ ← context[c_i]
11     return ⟨T(l), S⟩
12 Procedure evict(c_i, M_l, P*_l, S)
13     ⟨T, S, _⟩ ← context[c_i]; context[c_i] ← ⊥
14     ⟨T^c, S^c, H^c_pathMaps⟩ ← oramState
15     for sl ∈ T(l) do                    // update the tree
16         T*(l, sl) ← (T^c(l, sl) \ T(l, sl)) ∪ P*_l(sl)
17     S* ← (S^c \ S) ∪ {S}
18     H*_pathMaps ← H^c_pathMaps ∪ {M_l}
19     oramState ← ⟨T*, S*, H*_pathMaps⟩
```

When the server receives the request, it stores a reference to the current ORAM state in c_i's context until the client completes its access. Then, it returns the path map history $\mathcal{H}_{pathMaps}$, and the access sequence number seq (A2, L4-L8).

After receiving $\mathcal{H}_{pathMaps}$, c_i consolidates it into position map pm by invoking $consolidatePathMaps$ (first step of Fig. 4). This function applies the updates contained in $\mathcal{H}_{pathMaps}$ to the local position map, retaining for each block the location update with the highest timestamp. With the updated position map pm containing the most recent locations of the blocks, c_i discovers the current slot sl where b is stored. The first step terminates with c_i randomly choosing a path $\mathcal{P}_l$ that contains sl (A1, L5).

Second step (A1, L6-L12). Next, c_i will retrieve b from the server and read/write its content. Since b can either be in the tree or stash, c_i fetches the multi-version path $\mathcal{P}_l$ and stashes from the server by invoking Server.getPS.

The server processes c_i's request (A2, L9-11) by retrieving the ORAM state from c_i's context. Then, it collects path $\mathcal{P}_l$ from $\mathcal{T}$ and returns it along with stashes $\mathcal{S}$. By using the tree and stashes from c_i's context, the server ensures the tree is consistent with the pm consolidated in the previous step.

When c_i receives the response of Server.getPS, it merges the multiple versions of the blocks received in $\mathcal{P}_l$ and $\mathcal{S}$ into a working set W by invoking $mergePathStashes$ (second step of Fig. 4). This function uses the consolidated pm as a reference to filter blocks by retaining those with timestamps contained in pm, i.e., the more recent versions. From W, c_i retrieves b and updates its content and version if the operation is of type *write* (A1, L8-12).

Third step (A1, L13-L15). In the last step, c_i will evict blocks from W back to the server in a new path and stash. The redistribution of blocks must ensure two fundamental properties for MVP-ORAM: (1) the stash's expected size is bounded and proportional to the number of concurrent clients, and (2) the most accessed blocks are expected to be either in the highest levels of the tree or in the stash, giving more path options for clients to access them.

Eviction in detail. This is achieved by the $populatePath$ auxiliary function (third step of Fig. 4). First, c_i constructs a new path $\mathcal{P}_l^*$ by placing blocks from W into their correct slots according to pm. If multiple blocks are assigned to the same slot due to concurrent accesses, then the one with the highest timestamp is placed in the path, while the rest remain in the working set.

Then, c_i exchanges Z blocks from the working set with up to Z non-dummy blocks from random Z slots of $\mathcal{P}_l^*$, including the accessed block if it was in the path. This step ensures that the accessed block has the maximum number of paths available to retrieve it in the next access, while the expected stash size remains bounded. Note that the stash size can decrease if some of the Z selected slots are empty, since in this case, we remove blocks from the stash and add to these empty slots.

Next, c_i reorders blocks in $\mathcal{P}_l^*$ according to their timestamps, with more recently accessed blocks placed higher in the path, thus increasing the number of available paths for frequently accessed blocks.

Finally, c_i adds the remaining blocks in W, including the

accessed block, to a new stash S. It also updates the timestamp of blocks that were moved and update their locations on a new path map M_l. The function then returns P_l^*, S, and M_l.

After populating the path, c_i invokes Server.evict to send M_l, S, and $\mathcal{P}_l^*$ to the server (A1, L14). When the server receives an eviction request from the client, it first reads and cleans the client's context and obtains the current ORAM state (A2, L13-14). Then, it applies the modifications proposed by the client (A2, L15-19) by (1) updating path $\mathcal{P}_l$ in the current ORAM state, replacing the slots read by the client by the ones received in the eviction, (2) updating the set of stashes by replacing the retrieved stashes with the new stash, and (3) adding the received path map to the path map history.[3] These updated data structures are then stored in the ORAM state.

VI. Byzantine Fault-Tolerant ORAM

The previous section detailed MVP-ORAM, a protocol that can handle concurrent clients accessing an ORAM while satisfying wait-freedom and linearizability. We now describe how MVP-ORAM can be replicated using BFT SMR to tolerate fully malicious servers, ensuring data integrity and availability while preserving data and access-pattern secrecy. For this, we use a Byzantine Fault-Tolerant State Machine Replication (BFT SMR) protocol [1], [2].

BFT SMR is a classical technique for implementing fault-tolerant systems by replicating stateful, deterministic services on multiple fault-independent servers [1]. Most BFT SMR implementations allow tolerating t Byzantine faults with $n > 3t$ servers. This is possible by ensuring that each server starts in the same initial state and executes the same sequence of operations deterministically. Ensuring such a total order of operations on all correct servers requires executing Byzantine consensus [2], [81] to make the replicas agree on the next set of client operations to be executed.

MVP-ORAM solves three fundamental challenges that are required for replicating an ORAM through BFT SMR. First, it ensures the server-side algorithm is fully deterministic. Second, it requires only three invocations of state machine operations (getPM, getPS, and evict) for performing an access. Third, and most importantly, it makes ORAM accesses wait-free.

In detail, we execute n server replicas using a BFT SMR middleware (e.g., [82]) to ensure that the invocation of the three server operations used in Algorithm 1 is reliably disseminated in total order to all servers. Each server executes those functions locally, exactly as specified in Algorithm 2, and sends replies to the invoking clients, which consolidate a single response for each invocation by waiting for $t + 1$ matching replies.

Improving BFT ORAM performance. However, a direct implementation of MVP-ORAM in a BFT SMR system will significantly increase bandwidth usage, making the protocol highly inefficient. As such, we propose a series of optimizations that make the BFT version of MVP-ORAM more practical. Most of these optimizations aim to decrease bandwidth requirements (analyzed in §VII-C).

The first optimization is related to the execution of consensus over metadata. Byzantine consensus protocols typically select one (the leader, as in PBFT [2]) or more (the DAG block proposers, as in Mysticeti [83]) proposers to disseminate batches of requests to be ordered. In such protocols, the client sends its request to all replicas, and proposers re-disseminate the request along with ordering information. However, if the clients' requests are large (as in our case, where evict must send a path, path map, and stash), the proposer's bandwidth will be easily exhausted. To solve this, the client can send the evict parameters directly to the servers without ordering them and send only their hash for ordering through BFT SMR. Using the hashes, replicas retrieve the operation parameters and process the request as usual, thus significantly reducing the best-case bandwidth usage.

The second optimization aims to decrease the bandwidth usage of server responses. In traditional BFT-SMR, all correct servers respond with the result of executing the client-issued operation. This negatively affects bandwidth usage, especially during getPS when servers send multiple paths and stashes. We reduce this impact by employing an optimization introduced in PBFT [2] in which, for each ordered request, we choose a server that responds with the full reply while others respond with its cryptographic hash.

The client randomly selects a server that will send a full reply and accepts the response when the hash of this reply matches t hashes sent by other servers. If the obtained response does not match the hashes, the client asks t servers to send the full content.

Encryption keys management. ORAM protocols typically assume that clients manage and coordinate the shared cryptographic keys needed to encrypt the stored data or that there is a trusted third party (e.g., a proxy) that manages those keys. In MVP-ORAM, we remove this assumption through the use of secret sharing [10], more specifically, Dynamic Proactive Secret Sharing (DPSS), which is more appropriate for practical confidential BFT SMR systems [9].

When the servers are set up, the client generates a new random cryptographic key and secretly shares it, sending a different share to each server. The servers keep this share as part of their internal state. Then, when a client starts a new access and invokes Server.getPM, the servers send their stored shares along with the response. The client reconstructs the key using the received shares and uses it in all cryptographic operations during an access.

For simplicity, we assume the same key is used to encrypt all ORAM data. However, using this approach, we could have different keys for different data structures or even for different versions of the same data structures.

[3] To prevent unlimited growth of the history, clients send the consolidated position map every γ accesses, deleting outdated path maps.

VII. Security and Complexity Analysis

We now discuss the security, correctness, and theoretical performance of MVP-ORAM.

A. Security Analysis

The security of MVP-ORAM is defined by Theorem 1.

Theorem 1. *Given an ORAM of size N with c concurrent clients issuing requests from a distribution of accesses $\mathcal{D}$, then MVP-ORAM is an $\mu(N, c, \mathcal{D})$-secure Asynchronous Wait-Free ORAM as per Definition 1.*

To prove this theorem, we must show that MVP-ORAM fulfills both the *Correctness* and *Obliviousness* properties of Definition 1. We next outline these proofs, leaving their complete versions for the extended version of this paper [84].[4]

Correctness. MVP-ORAM provides an abstraction of a memory that can be written and read through the access operation without revealing to the server which memory/block address was accessed. Therefore, from the distributed computing point of view, we have to prove our construction implements N safe and live atomic read/write registers [85]. This requires proving all memory operations finish (wait-freedom [36]) and that they are safe under concurrent accesses (Linearizability [42]).

We prove *safety* by first showing that all access operations preserve the most up-to-date version of each accessed block on evictions (Lemma 3). This is important because it enables us to prove the safety of each logical block individually.

Lemma 3 (State Preservation)**.** *With the exception of the block b accessed during a write, an execution of* access *preserves the state of the ORAM.*

To prove safety under concurrent access of a single block, we have to prove each concurrent history of operations invoked on MVP-ORAM satisfies linearizability [42]. To prove this, we present a series of transformations of the observed history (respecting the MVP-ORAM algorithms) until we prove that the resulting high-level history containing only read and write operations is linearizable (Theorem 2). As part of this proof, we show that every read of a block returns the value written in the closest preceding write.

Theorem 2 (Linearizability)**.** *For each memory position b, MVP-ORAM's read (*access$(_, read, b, \perp)$*) and write (*access$(_, write, b, _)$*) operations satisfy linearizability.*

We prove that MVP-ORAM guarantees *wait-freedom* by showing that every step of the MVP-ORAM protocol terminates, assuming that the underlying BFT SMR guarantees liveness. Hence, every invocation of *acccess* by a correct client terminates (Theorem 3).

Theorem 3 (Wait-freedom)**.** *Every invocation of MVP-ORAM's* access *by a correct client terminates.*

[4]The numbering of theorems and lemmas referenced in this paper is the same as in the extended version.

Obliviousness. For this analysis, we assume $\mathcal{D}$ follows a Zipfian distribution [69], meaning that the frequency $f(r)$ of accessing the r^{th} most frequently accessed block (rank r) decreases proportionally to $r^{-\alpha}$. For example, when $\alpha = 1$, 27% of the blocks are accessed much more frequently than the others, with their access frequencies decreasing as their rank increases.

In Path ORAM, each block is mapped to a specific path in the tree, and when a client accesses a block, it retrieves the entire path and randomly re-assigns the block to a new path before eviction. In contrast, MVP-ORAM allows a block to be accessed through any path that contains it. Besides, the block is not reassigned to a new path after access; instead, it remains in the stash until it is evicted to the path in a future access to a different block. The next path used to request the block is only determined when it is accessed again. Additionally, only Z random blocks are evicted at a time from the stash, and instead of blocks being randomly placed in a path, they are sorted so that more frequently accessed blocks (i.e., with a higher timestamp) are placed up in the tree, giving them more possible paths for future requests.

Given this, we analyze MVP-ORAM's security in three different scenarios: (1) a single client accesses the ORAM once per timestep, (2) multiple clients access different blocks per timestep, and (3) multiple clients access the same block in the same timestep.

When a sequence of requests $\overrightarrow{y}$ is performed, the servers see $A(\overrightarrow{y})$, which is the same sequence of requests but transformed by the ORAM. When c clients access different blocks within the same timestep (case 2), each of them selects a random leaf, resembling the behavior of a single client performing c accesses across c consecutive timesteps (case 1). In Lemma 7 we show that $A(\overrightarrow{y})$ becomes indistinguishable from a random sequence of requests with high probability in case 1 (case 2 is ommitted, since they are similar).

Lemma 7. *When a single client accesses the ORAM per timestep, the access pattern $A(\overrightarrow{y})$ observed by the server during a sequence of requests $\overrightarrow{y}$ is computationally indistinguishable from a random sequence with high probability.*

For the last case, we must show that the access pattern generated by MVP-ORAM might be distinguishable from a random access pattern, particularly for blocks located near the leaves. We establish this result by computing the statistical distance [86] between the access pattern generated by MVP-ORAM and a random access pattern (Theorem 4). The intuition behind computing such a distance is as follows. In a random sequence of size c, we expect to observe c distinct leaves being accessed. However, when c clients simultaneously access the same block — particularly if the block is located near the leaves — the expected number of distinct leaves involved may be less than c. As such, we compare the distribution of the number of distinct leaves in a random sequence with the distribution of the number of distinct leaves generated in the worst-case execution of MVP-ORAM.

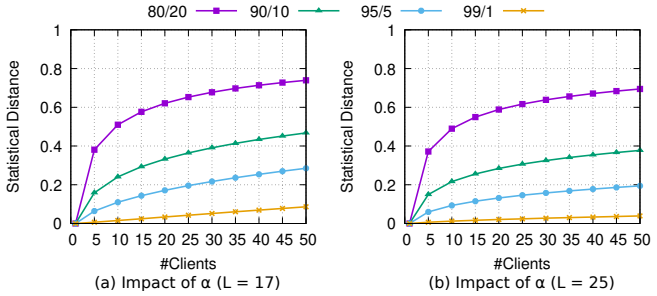

Fig. 5. Statistical distance simulation for different heights (L) and percentage of frequently accessed blocks (i.e., Zipfian parameter α). $80/20$ means 20% of blocks are accessed with probability 80% (i.e., $\alpha = 0.90$ in (a) and $\alpha = 0.88$ in (b)).

Theorem 4. *Given $c, N \in \mathbb{N}$, $\alpha \in \mathbb{R}$, and $D \in \mathcal{U}$, the statistical distance between a random sequence of size c and the access pattern generated by MVP-ORAM is bounded by $\mu(N, c, D(\alpha))$.*

Fig. 5 shows the statistical distance of the distributions (a worst-case measurement), considering different values of L and α. As expected, the distance decreases as we make the accesses more skewed (i.e., as we increase α) and as we increase the tree size L. If we decrease α to a point where the number of frequently accessed blocks approximates N (i.e., blocks are uniformly accessed), the statistical distance will be near 1 (worst security). Nonetheless, it is worth noting that although the statistical security in this case is far from good, the concrete probabilities of leakage are very small. The expected probability of c clients accessing the same block using the same path with uniform access distribution is $\frac{1}{2N^{c-1}}$. For example, this probability for two clients and a tree of $N = 2^{18}$ is less than 0.0002%.

B. Stash Size Analysis

The performance of MVP-ORAM is directly tied to its stash size, with larger stash sizes leading to reduced performance. Hence, it is crucial to ensure that the size of the stash does not grow indefinitely. Here, we outline the stash size analysis, with the complete proofs presented in the extended version [84].

Recall that the adversary can control the number of concurrent clients in MVP-ORAM through network scheduling. We argue that the adversary can maximize the stash size by maximizing concurrency. In other words, the largest stash size occurs when, at each timestep, the maximum number of clients c concurrently access the ORAM.

To analyze the stash size under this worst-case scenario, we show that concurrent clients add approximately cZ blocks to the stash in each timestep. When the stash size is small, clients may be unable to remove cZ blocks due to overlaps caused by multiple clients selecting the same blocks. However, as the stash size grows to $O(c \log N)$, the system reaches a point where concurrent clients can remove approximately cZ blocks from the stash, being Z a small constant. At this point, the stash size stabilizes as the rate of blocks being added to the stash approximately matches the rate of blocks being removed. We formalize this result in Theorem 5.

TABLE II MVP-ORAM COMMUNICATION COMPLEXITY WHEN STORING N BLOCKS USING n SERVERS WITH c ACTIVE CLIENTS.

	n servers with optimizations		
Operation	**Request**	**Response**	**Total Operation**
getPM	$O(n)$	$O(n + c(c + \log N))$	$O(n + c(c + \log N))$
getPS	$O(n)$	$O(n + c^2 \log N)$	$O(n + c^2 \log N)$
evict	$O(nc \log N)$	$O(n)$	$O(nc \log N)$

Theorem 5. *Under the worst-case scenario concerning concurrency, the expected stash size at any timestep is $O(c \log N)$.*

C. Bandwidth and Storage Analysis

We now analyze the communication complexity of MVP-ORAM. To simplify this analysis, we omit the cost of sending constant values such as block and client ids. We begin by considering a single-server (non-BFT) setup. The getPM reply contains up to c path maps of size $O(c + \log N)$, resulting in $O(c(c + \log N))$ bits. The getPS reply has size $O(c^2 \log N)$, as it contains c paths and stashes. Finally, an evict has size $O(c \log N)$ (a consolidated path and stash).

When considering an n-server setup, by default, the communication goes up by at least a factor of n, as data must be sent to all servers. Note that this multiplicative factor only occurs in requests, as replies benefit from the optimization of $n - 1$ servers sending hashes; i.e., $O(|Reply| + n)$. Requests also account for the cost of Byzantine consensus, which we use only for metadata ordering, which results in $O(n|Request| + Consensus)$.

By using a linear consensus protocol (e.g., HotStuff [81]) or a quadratic protocol [2], [82] in a small group of replicas (i.e., $n \ll c \log N$), the $Consensus$ term loses importance, and the complexity boils down to the values of Table II.

Putting it all together, MVP-ORAM incurs a communication complexity of $O((n + c)c \log N)$. This shows two nice properties of our protocol. First, it is an *adaptive wait-free construction* [46], meaning that its performance depends on the number of active concurrent clients c, not on the total number of existing clients. Second, its bandwidth usage is linearly proportional to the number of servers. This means that, with low concurrency, the bandwidth usage approximates that of Path ORAM replicated to n servers.

In terms of storage, each server needs to store the original position map ($O(N)$), the tree database ($O(N)$), and stash ($O(c \log N)$) plus up to c updates performed by different clients ($O(c^2 \log N)$) and not yet consolidated in the database. This leads to $O(N + c^2 \log N)$ server storage requirement.

VIII. STRONG MULTI-VERSION PATH ORAM

MVP-ORAM gives the same guarantees as non-wait-free ORAMs when a single client accesses the ORAM per timestep, or multiple clients concurrently access distinct addresses. However, obliviousness can be compromised if clients try to access the same block in the same timestep, particularly if this block is deep in the tree. In the worst-case scenario

(ORAM accesses follow a uniform distribution), clients will have few paths available to access the blocks since most of them will be located in slots near the leaves. In this case, the adversary can potentially observe concurrent clients requesting the same paths, compromising obliviousness.

In this section, we outline a variant of MVP-ORAM that preserves wait-freedom and obliviousness, at the cost of executing extra dummy requests for each real access and assuming the relative speed of clients in executing an access is approximately the same, i.e., no client is significantly faster or slower than the others. A full description of the Strong MVP-ORAM protocol and its proof is presented in the extended version of the paper [84].

A. Mitigating the Risk of Concurrent Accesses

Let $\sigma \geq 0$ be a security parameter defining the number of dummy accesses sent for each real access. To *access* a block b with address $addr$, each client implicitly builds a schedule of MVP-ORAM accesses (with the three steps described before) for $\sigma + 1$ consecutive timesteps. The key idea is to make at most one client access $addr$ in a timestep, while others execute dummy accesses. Let $\tau_i \in \{0, \ldots, \sigma\}$ be the timestep when c_i performs the real access. The schedule of c_i is built as follows: c_i first executes τ_i dummy accesses, then its real access, and concludes with $\sigma - \tau_i$ dummy accesses.

For this strategy to be effective, concurrent clients must define distinct timesteps for their actual accesses. A client c_i discovers τ_i during its first MVP-ORAM access. In the invocation of getPM, each client additionally sends the (encrypted) $addr$ that it wants to access, and the server stores this information in a set $\mathcal{A}$, which is returned together with the getPM's response.

When c_i receives $\mathcal{A}$, it defines τ_i by counting the number of ongoing accesses to $addr$, i.e., it sets τ_i as the number of accesses to $addr$ in $\mathcal{A}$ minus one (to ignore its own access). For example, if there is one access to $addr$ in $\mathcal{A}$ (it's own access, just declared), then $\tau_i = 0$ (the first access will retrieve the target block, and all the other σ will be dummy). If there are two accesses, then $\tau_i = 1$, which implies a single dummy access, the real access, and $\sigma - 1$ dummy accesses.

B. Security Analysis of Strong MVP-ORAM

Our strategy requires every client to always perform $\sigma + 1$ accesses, ensuring that in case concurrent clients access the same address, they do so at different timesteps. Since the server does not know which address a client is accessing, the adversary will be unable to distinguish between real and dummy accesses and whether clients are coordinating to hide access to the same address. Hence, as long as σ matches the maximum concurrency of the system ($c \leq \sigma + 1$), then this approach ensures no two clients access the same block in the same timestep and it is possible to show that the access pattern will be indistinguishable from a random access pattern, matching the obliviousness of parallel ORAM without giving up wait-freedom but assuming synchrony.

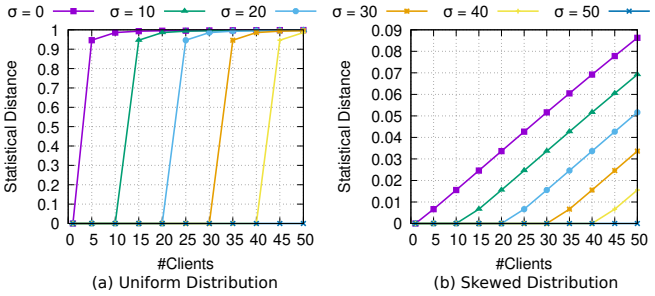

Fig. 6. Statistical distance simulation for different numbers of dummy accesses considering uniform ($\alpha << 1$) and skewed ($\alpha >> 1$) block selection distributions in a tree of height 17. In (a), clients access 99% of blocks with probability 99%. In (b), clients access 1% of blocks with probability 99%.

Fig. 6 shows how different values of σ affect the statistical distance between access distributions. The distance is zero (statistical security) when the number of clients c is at most $\sigma + 1$. However, when σ is less than c, the distance increases as multiple clients might access the same block in the same timestep. Nevertheless, it shows that our stronger variant improves statistical distance even when block selection follows a near-uniform distribution, i.e., most accessed blocks are near leaves. Specifically, it reduces the statistical distance from near 1 to zero when $c \leq \sigma + 1$. In this condition, MVP-ORAM's obliviousness approximate that of collision-free parallel ORAM [19] (Theorem 6).

Theorem 6. *Given $c, \sigma, N \in \mathbb{N}$, if $c \leq \sigma+1$, then Strong MVP-ORAM's access pattern is indistinguishable from a random access pattern with negligible probability in N.*

IX. Implementation & Evaluation

We implemented a prototype of MVP-ORAM and conducted a set of experiments on AWS to evaluate its stash size and concrete performance under different configurations.

A. Implementation

We built a prototype of MVP-ORAM in Java by extending COBRA [9], a confidential BFT SMR framework based on DPSS. COBRA itself relies on BFT-SMaRt [82], a replication library that provides all the features required for practical BFT SMR systems. BFT-SMaRt implements a Verifiable and Provable Consensus [87] based on Cachin's Byzantine Paxos [88], which is similar to PBFT [2], i.e., it requires three communication steps and has a quadratic message complexity in the common case. This is the consensus algorithm executed during the invocation of Server.getPM, Server.getPS, and Server.evict. Thus, through COBRA (and BFT-SMaRt), we can easily implement all features required by MVP-ORAM.

We have also implemented a safeguard against an unbounded number of concurrent clients exhausting bandwidth and storage by making ORAM servers only allow c_{max} clients to perform concurrent accesses. This is important to avoid memory trashing and ensure the stability of the system under

high load. Given enough resources, c_{max} could naturally match the number of clients accessing the shared ORAM.

Our implementation and all the code used for the experiments are available on the project's web page [47].

B. Setup and Methodology

Our experimental evaluation was performed in the AWS cloud using two types of instances. The servers were executed in n *r5n.2xlarge* instances, each having 8 vCPU, 64 GB of RAM, and 8.1 Gbps baseline network bandwidth. The clients were executed in 6 *c5n.2xlarge* instances, each with 8 vCPU, 21 GB of RAM, and 10 Gbps baseline network bandwidth.

We installed Ubuntu Server 22.04 LTS and OpenJDK 11 on all of the machines. The experiment analyzing stash size was conducted by simulating concurrent accesses to ORAM on a single machine. Performance was measured by executing servers in n machines and clients in the remaining ones. Throughput and latency measurements were collected from a single server and client machine, respectively. Unless stated otherwise, we measure MVP-ORAM considering a database of 1 GB configured in a tree of height $L = 17$ and with each node containing $Z = 4$ blocks of 4096 bytes each. Therefore, our database contains $N = 262143$ blocks. For each operation, clients randomly choose among all the blocks in a Zipfian distribution with $\alpha = 1.0$, and the operation type is picked uniformly at random between *read* and *write*. The graphs showing throughput and latency display the average (plus standard deviation) of data collected over 6 minutes for each experiment.

C. MVP-ORAM Stash Size

We start by studying the impact of the number of frequently accessed blocks and concurrent clients on the stash usage. Since the stash size changes over time with the number of accesses and does not depend on the size of the blocks, we run local simulations on a single server with a small block size to execute $500k$ concurrent accesses.

Since altering α affects the number of frequently accessed addresses and c affects concurrency, this experiment analyzes the impact of changing those parameters. The bucket size Z also affects the stash usage. However, it has the expected result: the stash size decreases as we increase Z since the probability of selecting empty slots from the path increases, which increases the number of blocks evicted from the stash.

The results are presented in Fig. 7. There are two main takeaways from these experiments: (1) the stash size stabilizes after some operations, and (2) the maximum stash size increases as we decrease α and increase c.

As analyzed in §VII-B, the rate of moving blocks from the stash to the tree and vice-versa is equal when the stash size reaches its expected maximum size.

We study the impact of Zipfian exponent α by experimenting with low ($\alpha = 10^{-6}$), medium ($\alpha = 1.0$), and high ($\alpha = 2.0$) contention levels in the accessed blocks. Accordingly, clients approximately access 90%, 27%, and 0.001% of blocks, respectively, with a probability greater

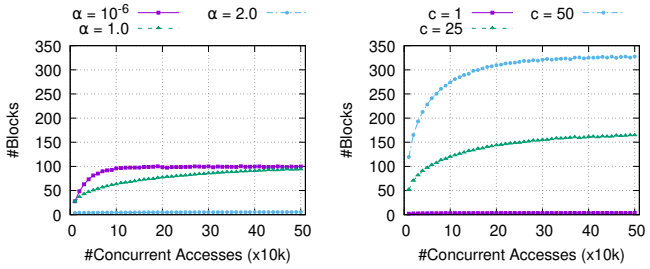

Fig. 7. Stash size with $c = 15$ and different Zipfian's exponent values, and $\alpha = 1.0$ and different number of clients. Each point represents an average of 50k accesses.

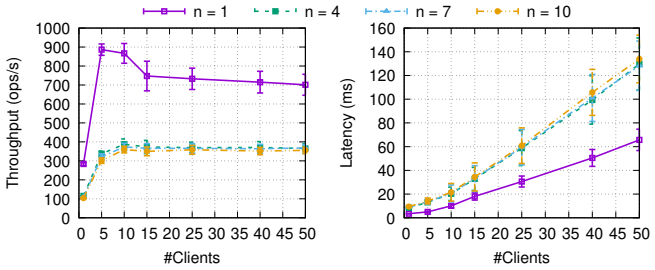

Fig. 8. MVP-ORAM throughput and latency for different numbers of replicas (n) and concurrent clients (c).

than 90%. Decreasing this parameter increases the number of distinct blocks that clients access and move to the stash. However, the number of concurrent clients heavily influences the number of distinct blocks swapped between the stash and the tree. Since c is fixed, the stash size is defined by the number of frequently accessed blocks, which is higher for small α values.

Increasing the number of concurrent clients also increases the stash size. Since blocks are uniformly sampled, increasing the number of clients increases the number of common blocks selected from the stash. Thus, the clients remove a few distinct blocks from the stash, increasing its size. This experiment shows that the number of concurrent clients heavily dominates the stash size, confirming our theoretical analysis (§VII-B).

D. MVP-ORAM Performance

The next set of experiments aims to measure the impact of BFT replication and the number of concurrent clients on MVP-ORAM's performance. Fig. 8 shows the throughput and latency of MVP-ORAM with varying n and c.

The overall throughput of the single-server setup is higher than that of the system tolerating failures. Specifically, the peak throughput is 2.2× the system's throughput tolerating one failure. Recall that a system tolerating t faults requires $n > 3t$ servers, and clients must wait for at least $t+1$ matching server responses before continuing. Therefore, as the number of tolerated faults increases, clients must send requests to more servers and wait for their responses, which increases latency and reduces the number of requests they send. In turn, this slightly reduces the overall throughput, i.e., it drops by 7% when the number of servers goes from 4 to 10.

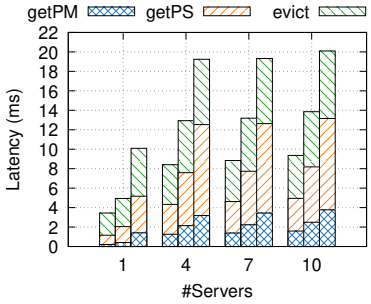

Fig. 9. Access latency breakdown. Each bar group considers runs with 1 (left), 5 (center), and 10 (right) concurrent clients.

TABLE III MVP-ORAM ACCESS REQUEST/RESPONSE SIZES IN BYTES, FOR WORKLOADS WITH 1, 5, AND 10 CLIENTS.

No. clients	getPM	getPS	evict
1	8/609	8/272415	307328/1
5	8/1489	8/445932	314101/1
10	8/2781	8/940107	339344/1

Although throughput remains stable regardless of the number of clients, latency increases as the number of clients increases. This is the effect of bounding the maximum number of concurrent accesses to $c_{max} = 10$ to avoid memory and bandwidth trashing.

To better understand the factors contributing to MVP-ORAM's performance, we break down the access latency in the three operations that constitute an access. Fig. 9 shows the individual latency of each protocol phase, which corresponds to an SMR operation for different numbers of servers and up to c_{max} clients, when queueing is not a factor. With a single client (no concurrency - first bar of each group), evict is the most costly phase. When the number of concurrent clients increases (second and third bars), getPS becomes more prominent. This can be explained by the operations' request/response sizes for different clients, as shown in Table III. Among the three phases of MVP-ORAM, getPS requires bandwidth that is quadratic in the number of clients (see Table II).

A note on the Strong MVP-ORAM performance. The strong variant of MVP-ORAM discussed in §VIII requires each ORAM access to perform $\sigma + 1$ MVP-ORAM accesses. This means the throughput observed for this variant would be MVP-ORAM throughput divided by $\sigma+1$. For example, if one wants to offer perfect ORAM guarantees for up to 10 clients, each access would require eleven MVP-ORAM accesses, which means a throughput of about the $\approx 390/11 = 35$ accesses/sec. A similar degradation also affects latency.

E. Performance with Different Configurations

To better understand how the performance of MVP-ORAM varies in different configurations, we conducted additional experiments for $n = 4$, considering various tree heights, bucket sizes, block sizes, and α values.

The first set of experiments considers different tree heights (L) and bucket sizes (Z). Fig. 10 shows the throughput and

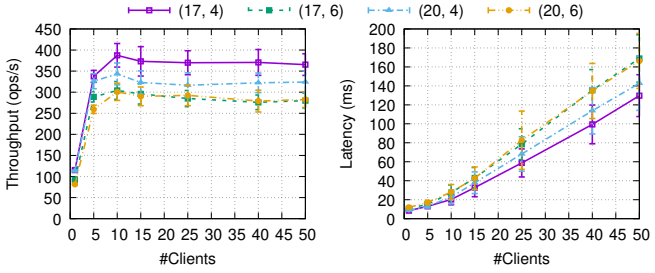

Fig. 10. MVP-ORAM throughput and latency for different tree heights and bucket sizes. (17, 4) means tree height 17 and bucket size 4.

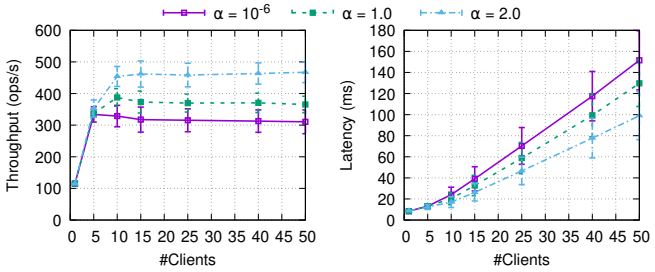

Fig. 11. MVP-ORAM throughput and latency for different values of α.

latency for different configurations (L, Z) of the database. Unsurprisingly, the results show that the performance gets worse when (L, Z) increases, since the amount of 4KB-blocks contained in a tree path is $L \times Z$.

The second set of experiments considers different values of α, the Zipfian distribution parameter used for selecting blocks to be accessed. Recall that smaller values of α make block accesses approximate a uniform distribution.

Fig. 11 shows the throughput and latency for the experiments. Although the effect of parameter α, which represents how skewed the system workload is, primarily reflects the obliviousness guarantee of the system, it also affects performance. This is due to the influence of skewness on the size of the stashes produced by clients (see §IX-C). The performance results show that when used in applications that induce skewed workloads, MVP-ORAM not only provides better security guarantees but also exhibits better performance.

Our final set of experiments evaluates the performance of MVP-ORAM with block sizes of 256 bytes, 1024 bytes, and 4096 bytes. Fig. 12 shows the throughput and latency for the experiments.

For small blocks of 256 bytes, the system reaches almost a thousand accesses per second. As the block size increases, performance decreases accordingly, since $4\times$ and $16\times$ more data is transferred with the other block sizes.

Overall, these experiments confirm our theoretical observation that MVP-ORAM performance gets worse in configurations that require more data to be transferred. This aligns with observations made for other ORAM protocols and supports the fundamental goal of enhancing the overall bandwidth of ORAM schemes.

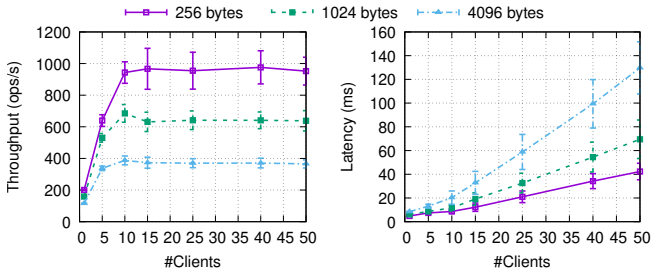

Fig. 12. MVP-ORAM throughput and latency for different block sizes.

TABLE IV PERFORMANCE COMPARISON OF MVP-ORAM WITH COBRA AND QUORAM WITH 50 CLIENTS.

	$n = 4$		$n = 7$	
Protocol	**Throughput**	**Latency**	**Throughput**	**Latency**
COBRA	3767 ops/s	12 ms	3446 ops/s	13 ms
MVP-ORAM	356 ops/s	130 ms	355 ops/s	128 ms
QuORAM	183 ops/s	272 ms	163 ops/s	305 ms

F. Experimental comparison with other systems

Table IV compares MVP-ORAM performance with COBRA and QuORAM. COBRA [9] tolerates Byzantine faults and ensures Secrecy, but not Obliviousness. Since it accesses data by invoking a single request that leverages secret sharing, its throughput and latency are approximately an order of magnitude better than what was observed for MVP-ORAM. This illustrates the cost of adding Obliviousness to a BFT datastore without resorting to trusted components.

QuORAM [35] is the only replicated ORAM service we are aware of. It tolerates crash faults and uses trusted proxies, while MVP-ORAM tolerates Byzantine faults without requiring trusted components. The existence of trusted proxies collocated with servers in the same machine eliminates the need to execute bandwidth-hungry ORAM operations through the network, as they are executed only between the proxy (which acts as a single Path ORAM client) and the server. We confirmed the benefit QuORAM gained with this approach by executing it in the same setting as MVP-ORAM. With over 100 clients and for $n = 4$ and $n = 7$, QuORAM achieves a maximum throughput of around 1000 operations per second. However, our evaluation also shows that QuORAM performs much worse than MVP-ORAM with a restricted number of clients. As sumarized in Table IV, MVP-ORAM with 50 client can process 356 ($n = 4$) and 355 ($n = 7$) ops/s, while QuORAM only processes between 163 and 183 ops/s. The reason behind the low performance of QuORAM is its higher latency, which is caused by the use of proxies. In our experiments, MVP-ORAM's maximum latency is 130 ms, while QuORAM's minimum latency is 272 ms.

Notice that having too many clients increases the service attack surface, as ORAM clients must be mutually trusted. Therefore, we argue that achieving high performance with fewer clients is more important than achieving good numbers with 100 or more concurrent clients accessing the ORAM.

X. CONCLUSIONS AND FUTURE WORK

This paper presented MVP-ORAM, the first Byzantine fault-tolerant ORAM protocol. It enables fail-prone concurrent clients to access a shared data store without revealing any information about the accessed data or their access patterns. We show that in asynchronous networks, satisfying wait-freedom fundamentally compromises collision-freedom, affecting the security guarantees of our construction. To account for this, we propose a weaker security definition for asynchronous wait-free ORAMs, which may be secure enough for typical storage applications with skewed block accesses. We devise MVP-ORAM as a deterministic wait-free ORAM service and integrate it into a confidential BFT data store, which shows promising performance results. Additionally, we introduce a stronger variant of MVP-ORAM that ensures perfect access-pattern secrecy with additional assumptions.

This paper opens many avenues for future work. For example, it seems impossible to implement a perfect wait-free ORAM in asynchronous systems, but this remains to be proved. Furthermore, more efficient variants of MVP-ORAM can be devised to implement perfect obliviousness. Finally, state-of-the-art information dispersal [89] can potentially be used to improve the bandwidth requirements and concrete performance of replicated/BFT ORAM.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments, which helped improve the paper. We also thank Cristiano Santos for his initial work on BFT ORAM services, which sparked the results presented in this paper. This work was supported by FCT through the Ph.D. scholarship, ref. 2020.04412.BD, the SMaRtChain and APOSTLE projects, ref. 2022.08431.PTDC and 2023.12254.PEX, respectively, and the LASIGE Research Unit, ref. UID/00408/2025.

REFERENCES

[1] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.
[2] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Symposium on Operating Systems Design and Implementation*, 1999.
[3] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," http://bitcoin.org/bitcoin.pdf, 2008.
[4] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," Ethereum Foundation, Tech. Rep., 2014.
[5] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "DepSpace: A Byzantine Fault-Tolerant Coordination Service," in *European Conference on Computer Systems*, 2008.
[6] R. Padilha and F. Pedone, "Belisarius: BFT Storage with Confidentiality," in *International Symposium on Network Computing and Applications*, 2011.
[7] S. Basu, A. Tomescu, I. Abraham, D. Malkhi, M. K. Reiter, and E. G. Sirer, "Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols," in *Conference on Computer and Communications Security*, 2019.
[8] E. Kokoris-Kogias, E. C. Alp, L. Gasser, P. Jovanovic, E. Syta, and B. Ford, "CALYPSO: Private Data Management for Decentralized Ledgers," *VLDB Endowment*, vol. 14, no. 4, 2020.
[9] R. Vassantlal, E. Alchieri, B. Ferreira, and A. Bessani, "COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services," in *Symposium on Security and Privacy*, 2022.

[10] A. Shamir, "How to Share a Secret," *Communications of the ACM*, vol. 22, no. 11, 1979.

[11] G. R. Blakley, "Safeguarding Cryptographic Keys," in *International Worshop on Managing Requirements Knowledge*, 1979.

[12] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Inference Attack Against Encrypted Range Queries on Outsourced Databases," in *Conference on Data and Application Security and Privacy*, 2014.

[13] M. Naveed, S. Kamara, and C. V. Wright, "Inference Attacks on Property-Preserving Encrypted Databases," in *Conference on Computer and Communications Security*, 2015.

[14] Y. Zhang, J. Katz, and C. Papamanthou, "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption," in *USENIX Security Symposium*, 2016.

[15] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why Your Encrypted Database Is Not Secure," in *Workshop on Hot Topics in Operating Systems*, 2017.

[16] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *Symposium on Theory of Computing*, 1987.

[17] R. Ostrovsky, "Efficient Computation on Oblivious RAMs," in *Symposium on Theory of Computing*, 1990.

[18] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, 1996.

[19] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious Parallel RAM and Applications," in *Theory of Cryptography Conference*, 2016.

[20] B. Chen, H. Lin, and S. Tessaro, "Oblivious Parallel RAM: Improved Efficiency and Generic Constructions," in *Theory of Cryptography Conference*, 2016.

[21] T.-H. Hubert Chan and E. Shi, "Circuit OPRAM: Unifying Statistically and Computationally Secure ORAMs and OPRAMs," in *Theory of Cryptography Conference*, 2017.

[22] T.-H. H. Chan, K. Nayak, and E. Shi, "Perfectly Secure Oblivious Parallel RAM," in *Theory of Cryptography Conference*, 2018.

[23] T.-H. H. Chan, E. Shi, W.-K. Lin, and K. Nayak, "Perfectly Oblivious (Parallel) RAM Revisited, and Improved Constructions," in *Conference on Information-Theoretic Cryptography*, 2021.

[24] G. Asharov, I. Komargodski, W.-K. Lin, E. Peserico, and E. Shi, "Optimal Oblivious Parallel RAM," in *Symposium on Discrete Algorithms*, 2022.

[25] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation," in *Symposium on Discrete Algorithms*, 2012.

[26] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro, "TaoStore: Overcoming Asynchronicity in Oblivious Data Storage," in *Symposium on Security and Privacy*, 2016.

[27] E.-O. Blass, T. Mayberry, and G. Noubir, "Multi-client Oblivious RAM Secure Against Malicious Servers," in *Applied Cryptography and Network Security*, 2017.

[28] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Maliciously Secure Multi-Client ORAM," in *Applied Cryptography and Network Security*, 2017.

[29] A. Chakraborti and R. Sion, "ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM," in *Network and Distributed System Security Symposium*, 2019.

[30] E. Stefanov and E. Shi, "Multi-Cloud Oblivious Storage," in *Conference on Computer and Communications Security*, 2013.

[31] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S3ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing," in *Conference on Computer and Communications Security*, 2017.

[32] K. G. Larsen, M. Simkin, and K. Yeo, "Lower Bounds for Multi-server Oblivious RAMs," in *Theory of Cryptography Conference*, 2020.

[33] T. Hoang, A. A. Yavuz, and J. Guajardo, "A Multi-Server ORAM Framework with Constant Client Bandwidth Blowup," *Transactions on Privacy and Security*, vol. 23, no. 1, 2020.

[34] T. Hoang, J. Guajardo, and A. Yavuz, "MACAO: A Maliciously-Secure and Client-Efficient Active ORAM Framework," in *Network and Distributed System Security Symposium*, 2020.

[35] S. Maiyya, S. Ibrahim, C. Scarberry, D. Agrawal, A. E. Abbadi, H. Lin, S. Tessaro, and V. Zakhary, "QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore," in *USENIX Security Symposium*, 2022.

[36] M. Herlihy, "Wait-Free Synchronization," *Transactions on Programming Languages and Systems*, vol. 13, no. 1, 1991.

[37] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Annual Technical Conference*, 2010.

[38] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *Journal of the ACM*, vol. 65, no. 4, 2018.

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

[40] G. Asharov, I. Komargodski, and Y. Michelson, "FutORAMa: A Concretely Efficient Hierarchical Oblivious RAM," in *Conference on Computer and Communications Security*, 2023.

[41] K. G. Larsen and J. B. Nielsen, "Yes, There is an Oblivious RAM Lower Bound!" in *International Cryptology Conference*, 2018.

[42] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *Transactions on Programming Languages and Systems*, vol. 12, no. 3, 1990.

[43] M. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, no. 5, 2005.

[44] A. Dan, P. S. Yu, and J.-Y. Chung, "Characterization of Database Access Skew in a Transaction Processing Environment," *SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1, 1992.

[45] C. A. Lynch, "Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distribution of Column Values," in *International Conference on Very Large Data Bases*, 1988.

[46] Y. Afek, D. Dauber, and D. Touitou, "Wait-Free Made Fast," in *Symposium on Theory of Computing*, 1995.

[47] MVP-ORAM Team, "MVP-ORAM Open-Source Repository," https://github.com/rvassantlal/MVPORAM, 2025.

[48] J. S. Fraga and D. Powell, "A Fault-and Intrusion-Tolerant File System," in *International Conference on Computer Security*, 1985.

[49] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, "Responsive Security for Stored Data," *Transactions on Parallel and Distributed Systems*, vol. 14, no. 9, 2003.

[50] M. Marsh and F. Schneider, "CODEX: A Robust and Secure Secret Distribution System," *Transactions on Dependable and Secure Computing*, vol. 1, no. 1, 2004.

[51] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing*, vol. 11, no. 4, 1998.

[52] N. Mohnblatt, A. Sonnino, K. Gurkan, and P. Jovanovic, "Arke: Scalable and Byzantine Fault Tolerant Privacy-Preserving Contact Discovery," in *Conference on Computer and Communications Security*, 2024.

[53] R. Ostrovsky and V. Shoup, "Private Information Storage (Extended Abstract)," in *Symposium on Theory of Computing*, 1997.

[54] P. Williams, R. Sion, and B. Carbunar, "Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage," in *Conference on Computer and Communications Security*, 2008.

[55] B. Pinkas and T. Reinman, "Oblivious RAM Revisited," in *International Cryptology Conference*, 2010.

[56] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly Secure Oblivious RAM without Random Oracles," in *Theory of Cryptography Conference*, 2011.

[57] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM," in *Theory of Cryptography Conference*, 2016.

[58] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious Memory Primitives from Intel SGX," in *Network and Distributed System Security Symposium*, 2018.

[59] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX," in *Network and Distributed System Security Symposium*, 2018.

[60] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "PanORAMa: Oblivious RAM with Logarithmic Overhead," in *Symposium on Foundations of Computer Science*, 2018.

[61] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal Oblivious RAM," *Journal of the ACM*, vol. 70, no. 1, 2022.

[62] N. Crooks, M. Burke, E. Cecchetti, S. Harel, R. Agarwal, and L. Alvisi, "Obladi: Oblivious Serializable Transactions in the Cloud," in *Symposium on Operating Systems Design and Implementation*, 2018.

[63] W. Cheng, D. Sang, L. Zeng, Y. Wang, and A. Brinkmann, "Tianji: Securing a Practical Asynchronous Multi-User ORAM," *Transactions on Dependable and Secure Computing*, vol. 20, no. 6, 2023.

[64] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Journal of the ACM*, vol. 42, no. 1, 1995.

[65] I. A. Escobar, E. Alchieri, F. L. Dotti, and F. Pedone, "Boosting Concurrency in Parallel State Machine Replication," in *International Middleware Conference*, 2019.

[66] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, vol. 35, no. 2, 1988.

[67] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous BFT Made Practical," in *Conference on Computer and Communications Security*, 2018.

[68] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-NG: Fast Asynchronous BFT Consensus with Throughput-Oblivious Latency," in *Conference on Computer and Communications Security*, 2022.

[69] G. Kingsley Zipf, *Selected Studies of the Principle of Relative Frequency in Language:*. Harvard University Press, 1932.

[70] A. van Renen, D. Horn, P. Pfeil, K. Vaidya, W. Dong, M. Narayanaswamy, Z. Liu, G. Saxena, A. Kipf, and T. Kraska, "Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet," *VLDB Endowment*, vol. 17, no. 11, 2024.

[71] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," in *Annual Technical Conference*, 2008.

[72] S. T. Leutenegger and D. Dias, "A Modeling Study of the TPC-C Benchmark," in *International Conference on Management of Data*, 1993.

[73] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Symposium on Cloud Computing*, 2010.

[74] P. Grubbs, A. Khandelwal, M.-S. Lacharité, L. Brown, L. Li, R. Agarwal, and T. Ristenpart, "PANCAKE: Frequency Smoothing for Encrypted Data Stores," in *USENIX Security Symposium*, 2020.

[75] S. Maiyya, S. C. Vemula, D. Agrawal, A. El Abbadi, and F. Kerschbaum, "Waffle: An Online Oblivious Datastore for Protecting Data Access Patterns," *Management of Data*, vol. 1, no. 4, 2023.

[76] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov, "Anonymous RAM," in *European Symposium on Research in Computer Security*, 2016.

[77] W. Chen and R. A. Popa, "Metal: A Metadata-Hiding File-Sharing System," in *Network and Distributed System Security Symposium*, 2020.

[78] W. Chen, T. Hoang, J. Guajardo, and A. A. Yavuz, "Titanium: A Metadata-Hiding File-Sharing System with Malicious Security," in *Network and Distributed System Security Symposium*, 2022.

[79] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the Depth of Oblivious Parallel RAM," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[80] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *International Conference on Distributed Computing Systems*, 2003.

[81] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT Consensus with Linearity and Responsiveness," in *Symposium on Principles of Distributed Computing*, 2019.

[82] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt," in *International Conference on Dependable Systems and Networks*, 2014.

[83] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian, "Mysticeti: Reaching the Latency Limits with Uncertified DAGs," in *Network and Distributed System Security Symposium*, 2025.

[84] R. Vassantlal, H. Heydari, B. Ferreira, and A. Bessani, "MVP-ORAM: a Wait-free Concurrent ORAM for Confidential BFT Storage," https://doi.org/10.48550/arXiv.2512.12006, 2025.

[85] L. Lamport, "On Interprocess Communication: Part I: Basic Formalism," *Distributed Computing*, vol. 1, no. 2, 1986.

[86] L. Reyzin, "Extractors and the leftover hash lemma," 2011.

[87] J. Sousa and A. Bessani, "From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation," in *European Dependable Computing Conference*, 2012.

[88] C. Cachin, "Yet Another Visit to Paxos," IBM Research Zurich, Tech. Rep. RZ 3754, 2009.

[89] N. Alhaddad, S. Das, S. Duan, L. Ren, M. Varia, Z. Xiang, and H. Zhang, "Brief Announcement: Asynchronous Verifiable Information Dispersal with Near-Optimal Communication," in *Symposium on Principles of Distributed Computing*, 2022.

APPENDIX A
MVP-ORAM AUXILIARY FUNCTIONS

Algorithm 3 specifies the auxiliary functions used by clients to update their position map (consolidatePathMaps), merge multiple versions of path and stashes (mergePathStashes), and to create a new version of path and stash (populatePath), as used in Algorithm 1 (§V).

The *consolidatePathMaps* function (A3, L1-7) receives as input a history of path maps containing the location and version updates of blocks that have been evicted until the access that invoked this function. Using this history, it updates the local position map by keeping for each block address the location update with the highest timestamp, while discarding location updates with older versions.

The *mergePathStashes* function (A3, L8-11) receives as input a multi-version path, a set of stashes, and a consolidated position map. Using the position map as a reference point, the function filters blocks received in the path and stashes by keeping the ones with received in the correct slot and with timestamp according to the position map. Older and duplicated blocks are ignored.

Finally, the *populatePath* function (A3, L12-39) receives a working set, a path identification, accessed address, consolidate position map, and the sequence number of the access as input and populates a new path and stash with blocks contained in the working set. This is achieved in four steps on the *populatePath* auxiliary function. First (A3, L15-20), c_i populates the new path $\mathcal{P}_l^*$ by putting blocks from W in their correct slots according to pm. If multiple clients have evicted different blocks to the same slot during the previous concurrent accesses, then c_i selects the block with the highest sequence among them and keeps others in W (A3, L17). Additionally, c_i keeps track of non-empty slots in S_{used}.

In the second step (A3, L21-25), c_i exchanges Z blocks from the stash with Z blocks from $\mathcal{P}_l^*$. This is done by sampling Z slots from $\mathcal{P}_l^*$, including the accessed block if it was not previously in the stash, ensuring that the maximum number of paths is available to retrieve this block in the next accesses. Then, c_i samples Z blocks from W uniformly at random, except the accessed block, and puts them in set B_Z. After selecting Z blocks and slots, c_i iterates over selected slots. For each slot sl^*, if it contains a real block, it is moved to W, and a block from B_Z is moved to sl^*. Note that the stash size can decrease if some of the selected slots are empty, as we remove blocks from the stash rather than substituting them.

During the third step (A3, L26-31), c_i reorders blocks by placing recently accessed blocks higher in the path. It first collects all blocks from $\mathcal{P}_l^*$ into B_l. Then, iterates over each slot in the path after the exchange (i.e., $S_{used} \cup S_Z$) *from the lowest slot to the highest* and evicts block with highest access among B_l to it. Additionally, c_i updates the timestamp of the evicted blocks and their location on the path map M_l.

Finally, during the fourth step (A3, L32-39), c_i builds the new stash S by adding the remaining blocks in W, updating the timestamp and location of newly added blocks to the stash, including the accessed block. The function then returns P_l^*, S, and M_l.

Algorithm 3: MVP-ORAM auxiliary functions.

1 **Function** *consolidatePathMaps*($\mathcal{H}_{pathMaps}$)
2 $\quad \forall addr = 0..N : pm[addr] \leftarrow \langle \perp, \langle -1, -1, -1 \rangle \rangle$
3 $\quad$ **foreach** $M_l \in \mathcal{H}_{pathMaps}$ **do**
4 $\quad\quad$ **foreach** $\langle addr, sl, ts \rangle \in M_l$ **do**
5 $\quad\quad\quad \langle \_, ts^* \rangle \leftarrow pm[addr]$
6 $\quad\quad\quad$ **if** $ts > ts^*$ **then** $pm[addr] \leftarrow \langle sl, ts \rangle$
7 $\quad$ **return** pm
8 **Function** *mergePathStashes*($\mathcal{P}_l, \mathcal{S}, pm$)
9 $\quad W_s \leftarrow \{\langle a, s, ts \rangle \in S : S \in \mathcal{S} \wedge pm[a] = \langle \perp, ts \rangle\}$
10 $\quad W_p \leftarrow \{\langle a, d, ts \rangle \in \mathcal{P}_l(sl) : sl \in \mathcal{P}_l \wedge pm[a] = \langle sl, ts \rangle\}$
11 $\quad$ **return** $W_s \cup W_p$
12 **Function** *populatePath*($W, l, addr, pm, seq$)
13 $\quad \mathcal{P}_l^*, S, M_l, S_{used} \leftarrow \emptyset$
14 $\quad \langle sl, \_ \rangle \leftarrow pm[addr]$
15 $\quad$ **foreach** $sl^* \in \mathcal{P}_l^*$ **do** `// put blocks on path l`
16 $\quad\quad B_{sl} \leftarrow \{\langle addr^*, s \rangle : \langle addr^*, \_, \langle \_, \_, s \rangle \rangle \in W \wedge$
$\quad\quad\quad pm[addr^*] = \langle sl^*, \_ \rangle\}$
17 $\quad\quad \langle addr', \_ \rangle \leftarrow$ entry with highest s from B_{sl}
18 $\quad\quad \mathcal{P}_l^*(sl^*) \leftarrow \{W[addr']\}$
19 $\quad\quad W \leftarrow W \setminus \{\mathcal{P}_l^*(sl^*)\}$
20 $\quad\quad S_{used} \leftarrow S_{used} \cup \{sl^*\}$
21 $\quad S_Z \leftarrow Z$ random slots from $\mathcal{P}_l^*$ including sl if $sl \neq \perp$
22 $\quad B_Z \leftarrow Z$ random blocks from $W \setminus \{\langle addr, \ ... \rangle\}$
23 $\quad$ **foreach** $sl^* \in S_Z$ **do** `// exchange Z blocks`
24 $\quad\quad$ **if** $\mathcal{P}_l^*(sl^*) \neq \perp$ **then** $W \leftarrow W \cup \mathcal{P}_l^*(sl^*)$
25 $\quad\quad \mathcal{P}_l^*(sl^*) \leftarrow$ set with a block from B_Z
26 $\quad B_l \leftarrow \{b \in \mathcal{P}_l^*(sl^*) : sl^* \in \mathcal{P}_l^* \wedge \mathcal{P}_l^*(sl^*) \neq \perp\}$
27 $\quad$ **foreach** $sl^* \in$ ***sort***$(S_{used} \cup S_Z)$ **do** `// reorder blocks`
28 $\quad\quad \langle addr^*, d, \langle v, a, s \rangle \rangle \leftarrow$ block with highest a from B_l
29 $\quad\quad B_l \leftarrow B_l \setminus \{\langle addr^*, d, \langle v, a, s \rangle \rangle\}$
30 $\quad\quad \mathcal{P}_l^*(sl^*) \leftarrow \{\langle addr^*, d, \langle v, a, seq \rangle \rangle\}$
31 $\quad\quad M_l \leftarrow M_l \cup \{\langle addr^*, sl^*, \langle v, a, seq \rangle \rangle\}$
32 $\quad$ **foreach** $\langle addr^*, b, \langle v, a, s \rangle \rangle \in W$ **do** `// build stash`
33 $\quad\quad \langle sl^*, \_ \rangle = pm[addr^*]$
34 $\quad\quad$ **if** $sl^* \neq \perp \vee addr^* = addr$ **then**
35 $\quad\quad\quad ts \leftarrow \langle v, a, seq \rangle$
36 $\quad\quad\quad M_l \leftarrow M_l \cup \{\langle addr^*, \perp, ts \rangle\}$
37 $\quad\quad$ **else** $ts \leftarrow \langle v, a, s \rangle$
38 $\quad\quad S \leftarrow S \cup \{\langle addr^*, b, ts \rangle\}$
39 $\quad$ **return** $\langle \mathcal{P}_l^*, S, M_l \rangle$

APPENDIX B
ARTIFACT APPENDIX

The artifact includes the implementation of MVP-ORAM, QuORAM,[5] and COBRA.[6] The former was used for performance assessment, while the latter two were used for comparison. The experiments were conducted on multiple AWS servers, but for ease of reproduction, we provide a scaled-down version of a single-machine evaluation setup in this appendix. We present instructions to reproduce the main results of our paper, which include the performance of MVP-ORAM and a comparison with QuORAM. Reproduction of other experiments is presented in the extended version of this paper [84].

A. Description & Requirements

1) How to access: The artifact is available on Zenodo,[7] which includes the MVP-ORAM, QuORAM, and COBRA implementations adapted for easier benchmarking.

2) Hardware dependencies: This artifact was tested on a machine with a 2.59 GHz CPU and 16 GB of RAM. Note that the experimental results presented in the paper were obtained using multiple more powerful machines instead of a single machine.

3) Software dependencies: Execution of this artifact requires *unzip*, *gnuplot*, and *OpenJDK 11*. It can be executed on either Linux or the Linux subsystem on Windows.[8]

4) Benchmarks: None.

B. Artifact Installation & Configuration

Download the artifact from Zenodo in the location where the experiments will be executed and open a terminal in that location. Let us designate the terminal as *builder* terminal during the experiments. Execute the following command in the *builder* terminal to extract the artifact:

```
unzip MVP-ORAM-Artifact.zip
```

After extracting, you should have a folder named `MVP-ORAM-Artifact` containing the artifact. Navigate to this folder in *builder* terminal and, **for all the remaining instructions, assume a relative path from it**.

The experiments are automated using a custom benchmarking tool, configured by setting parameters in the `config/benchmark.config` file.

C. Experiment Workflow

During the experiments, the collected and processed data will be stored in subfolders located at the path specified in `output.path`:

- `output/raw_data` will contain the raw data.
- `output/processed_data` will contain the processed data used for plotting.
- `output/plots` will contain the produced plots.

Hence, for all experiments, set `output.path` to the same location, e.g., the path to the `MVP-ORAM-Artifact` folder.

The artifact is divided into the MVP-ORAM and QuORAM projects. The experiment workflow of both projects is composed of the following steps:

1) Navigate to the project folder in *builder* terminal.
2) Build the project by executing the following command in *builder* terminal:

   ```
   ./gradlew localDeploy -PnWorkers=x
   ```

   where x is the number of servers plus the number of client workers. We recommend $x = 3 \times \texttt{max}(fault\_thresholds) + 3$. After a successful execution of the command, you should have $x + 1$ folders in `<project>/build/local/` named `controller` and `worker<i>`, where `i` $\in [0, \ldots, x-1]$. These folders contain all the necessary materials to execute the experiments.
3) Open a new terminal and navigate to `<project>/build/local/controller` folder. Let us designate this terminal as *controller* terminal.
4) Configure `benchmarking.config` located in the `<project>/build/local/controller/config` folder using a text editor (e.g., *nano*) for a given experiment.
5) Execute the following command in the *controller* terminal:

   ```
   ./smartrun.sh controller.
       BenchmarkControllerStartup config/
       benchmark.config
   ```

6) Execute the following command in the *builder* terminal:

   ```
   ./runscripts/startLocalWorkers.sh <x>
       127.0.0.1 12000
   ```

   The execution of this command will trigger the experiment in the *controller* terminal, and it will display the experiment status during the execution. The experiment concludes when the *controller* terminal prints the execution duration.

Tip: An active experiment can be terminated at any time by executing `CTRL+C` in the *controller* terminal.

D. Major Claims

We make the following claims in our paper:

- (C1): The stash size stabilizes after some number of accesses, and the maximum stash size increases as we decrease α and increase c. This is proven by experiment (E1), whose results are shown in Fig. 7.
- (C2): The overall throughput decreases and the latency increases as we increase the number of servers. This is demonstrated by experiment (E2), and reported in Fig. 8.
- (C3): The throughput stabilizes and the latency increases as we increase the number of clients. This is demonstrated by experiment (E2) and reported in Fig. 8.
- (C4): MVP-ORAM outperforms QuORAM both in throughput and latency with 50 clients. This is proven by experiment (E3), with results reported in Table IV.

[5] https://github.com/SeifIbrahim/QuORAM/
[6] https://github.com/bft-smart/cobra
[7] https://doi.org/10.5281/zenodo.17842154
[8] https://learn.microsoft.com/windows/wsl

E. Evaluation

1) Experiment (E1): [Stash] [10 human-minutes + 1.1 compute-hour]: This experiment shows the impact of α and c on the stash size. The results show that the stash size stabilizes over time, and its maximum size increases as α decreases (i.e., as the access distribution becomes more uniform) and c increases.

[Preparation] This experiment requires executing the steps described in the experimental workflow twice, with different values of `zipf_parameters` and `clients_per_round`. In both executions, use the following parameters:

- `global.worker.machines=3`
- `fault_thresholds=0`
- `tree_heights=16`
- `bucket_sizes=4`
- `block_sizes=8`
- `concurrent_clients=15`
- `measurement_duration=600`

First execution: Set `clients_per_round=5` and `zipf_parameters=0.000001 1.0 2.0`

Second execution: Set `clients_per_round=1 10 15` and `zipf_parameters=1.0`

[Execution] Consider the project `MVP-ORAM` and follow the steps described in the experimental workflow to execute the experiment twice, setting the parameters defined above.

[Results] Execute the following command, in the *builder* terminal, to produce Fig. 7:

```
gnuplot -e "O='<output.path>'; L='16'; Z='4';
    B='8'; c_max='15'; D=10" plotScripts/
    stash_plot.gp
```

The correct execution of this command will create `<output.path>/output/plots/stash.pdf` file containing the figure. Due to the scaled-down experiment, this figure only shows the overall trend of Fig. 7, confirming (C1).

Note: For more acurate results, increase `measurement_duration` and setting `D='d'` during plotting, where $d = \lceil measurement\_duration/60 \rceil$.

2) Experiment (E2): [Performance] [5 human-minutes + 0.5 compute-hour]: This experiment shows the impact of n servers and c clients on the throughput and latency of the system. It shows that increasing n decreases the throughput and increases the latency, and increasing c increases the latency while keeping the throughput stable.

[Preparation] Use the following parameters for this experiment:

- `global.worker.machines=12`
- `fault_thresholds=0 1 2`
- `clients_per_round=1 5 10 15`
- `tree_heights=16`
- `bucket_sizes=4`
- `block_sizes=8`
- `concurrent_clients=5`
- `measurement_duration=60`
- `zipf_parameters=1.0`

[Execution] Consider the project `MVP-ORAM` and follow the steps described in the experimental workflow to execute the experiment using the above values.

[Results] Execute the following command in the *builder* terminal to produce Fig. 8:

```
gnuplot -e "O='<output.path>'; L='16'; Z='4';
    B='8'; A='1.0'; c_max='5'" plotScripts/
    throughput_latency_plot.gp
```

The correct execution of this command will create the `<output.path>/output/plots/performance.pdf` file containing the figure. Due to the scaled-down experiment, this figure only shows the overall trend of Fig. 8, confirming (C2) and (C3).

3) Experiment (E3): [Comparison with QuORAM] [5 human-minutes + 1 compute-hour]: This experiment shows that MVP-ORAM outperforms QuORAM both in throughput and latency. However, due to the resources required to run experiments, we only show results for up to 15 clients.

[Preparation] Use the following parameters for this experiment:

- `global.worker.machines=7`
- `fault_thresholds=1 2`
- `clients_per_round=1 5 10 15`
- `storage.sizes=1`
- `bucket_sizes=4`
- `block_sizes=8`
- `measurement_duration=60`
- `zipf_parameters=1.0`

Note: QuORAM configuration file is located at `QuORAM/config/benchmark.config`.

[Execution] Consider the project `QuORAM` and follow the steps described in the experimental workflow to execute the experiment using the above values.

Note: Execute (E3) after completing (E2), as (E3)'s plot relies on (E2)'s results for comparison.

[Results] Run the following command in *builder* terminal:

```
gnuplot -e "O='<output.path>'; L='16'; Z='4';
    B='8'; A='1.0'; c_max='5'" plotScripts/
    mvp_oram_vs_quoram_plot.gp
```

The correct execution of this command will create the `<output.path>/output/plots/quoram.pdf` file containing a figure comparing MVP-ORAM with QuORAM. Due to the scaled-down experiment, this figure only shows the overall trend of Table IV, confirming (C4).

F. Customization

The experiments can be executed by setting different values for the used parameters, i.e., varying the fault threshold, tree height, bucket size, block size, Zipfian parameter, number of clients, and the maximum number of concurrent clients. However, to plot the measurements correctly, provide the correct values of the used parameters.

Additionally, the accuracy of the results can be improved by obtaining more data points by increasing the experiment duration, which can be achieved by modifying the `measurement_duration` parameter.