TaoStore: Overcoming Asynchronicity in Oblivious Data Storage

Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, Stefano Tessaro
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California, USA
{cetin, victorzakhary, amr, rachel.lin, tessaro}@cs.ucsb.edu

***Abstract*—We consider *oblivious* storage systems hiding both the contents of the data as well as access patterns from an untrusted cloud provider. We target a scenario where multiple users from a trusted group (e.g., corporate employees) asynchronously access and edit potentially overlapping data sets through a trusted proxy mediating client-cloud communication.**

The main contribution of our paper is twofold. Foremost, we initiate the first *formal* study of asynchronicity in oblivious storage systems. We provide security definitions for scenarios where both client requests and network communication are asynchronous (and in fact, even adversarially scheduled). While security issues in ObliviStore (Stefanov and Shi, S&P 2013) have recently been surfaced, our treatment shows that also CURIOUS (Bindschaedler at al., CCS 2015), proposed with the exact goal of preventing these attacks, is *insecure* under asynchronous scheduling of network communication.

Second, we develop and evaluate a new oblivious storage system, called Tree-based Asynchronous Oblivious Store, or TaoStore for short, which we prove secure in asynchronous environments. TaoStore is built on top of a new *tree-based* ORAM scheme that processes client requests concurrently and asynchronously in a non-blocking fashion. This results in a substantial gain in throughput, simplicity, and flexibility over previous systems.

I. Introduction

Outsourcing data to cloud storage has become increasingly popular and attractive. However, confidentiality concerns [9] make potential users skeptical about joining the cloud. Encryption alone is not sufficient to solve all privacy challenges. Typically, the *access patterns* are *not* hidden from the cloud provider, i.e., it can for example detect whether and when the same data item is accessed repeatedly, even though it does not learn what the item actually is. Data access patterns can leak sensitive information using prior knowledge, as shown e.g. in the setting of searchable symmetric encryption [24], [7].

This work targets *cloud storage* where multiple users from a trusted group (e.g., employees within the same company) need to access (in a read/write fashion) data sets which may overlap. To achieve this, users' accesses are mediated by a shared (trusted) proxy which coordinates these accesses and, at the same time, reduces the amount of information leaked to the cloud. *Oblivious RAM* (ORAM) – a cryptographic primitive originally proposed by Goldreich and Ostrovsky [17] for software protection – is the standard approach to make access patterns *oblivious*. Most ORAM solutions [38], [11], [10], [4] are not suitable for our multi-user scenario, as they handle operation requests *sequentially*, i.e., a new request is not processed until a prior ongoing request is completed, thus creating a bottleneck under concurrent loads. To date, only a handful of solutions leverage parallelism to increase throughput [36], [13], [44], [4]. PrivateFS [44] is based on hierarchical ORAM and supports parallel accesses from a limited number of clients. ObliviStore [36] (which is based on SSS-ORAM [37]) was the first work to consider the proxy model we also assume in this work. ObliviStore was recently revisited by Bindschaedler et al. [4], who proposed a new system called CURIOUS fixing a subtle (yet serious) security flaw arising in concurrent environments.

Our contributions, in a nutshell: Motivated by [4], this work initiates a comprehensive study of asynchronicity in oblivious storage. We make contributions along two axes:

1) We observe that the previous treatment has not captured crucial security issues related to asynchronicity in oblivious storage. We develop a comprehensive security framework, and present an attack showing that *access patterns in CURIOUS are not oblivious in an asynchronous environment* as captured by our model.
2) We design and evaluate a new provably secure system, called TaoStore, that fully resists attacks in asynchronous settings and also leverages the benefits of asynchronicity for better performance. Our system follows a completely different paradigm than previous works – in particular it departs from the SSS framework and is completely tree based – with substantial gains in simplicity, flexibility, and efficiency.

A. Asynchronicity vs Security

Asynchronicity is an important variable in the design of secure storage systems, and there are at least two ways in which it can affect them:

- *Asynchronous client requests.* Multiple client requests can come at any time point in time (either from the same client or from different ones), and should be answered independently of each other, possibly as soon as the data item is retrieved from the server in order not to slow down the applications requiring these accesses.
- *Asynchronous network communication.* The communication between the clients and the proxy, and the commu-

DOI 10.1109/SP.2016.20

nication between the proxy and the server, is in general asynchronous.

Needless to say, we would like our systems to be secure in such asynchronous environments. The first question we thus ask is:

Are existing systems secure under arbitrary scheduling of communication and operations?

The answer is negative: *all* existing approaches of handling concurrent requests on the *same* data item can leak substantial information under asynchronous scheduling. The authors of CURIOUS [4] have already shown that the sequentialization of accesses to the same block in ObliviStore renders the system insecure. We will go one step further, and show that CURIOUS itself has not completely resolved the issue, and is also insecure when operations are scheduled concurrently and communication is asynchronous.

Our attack assumes that the adversary learns the timings of the proxy's answers back to the client. We find this assumption reasonable. For example, the attacker may observe (encrypted) network traffic between the proxy and the clients, and moreover, a client may only schedule a new access (or perform some other noticeable action) when a previous access terminates. These timings were however kept secret in the original security definition of [36], also used in [4]. Therefore, our attack does not invalidate any of the claims from [36]. Still, it motivates us to develop a definitional security framework for asynchronous oblivious storage systems, which we believe to be of independent interest.

B. Asynchronicity vs Efficiency

Our security assessment calls for a system which is fully secure in an asynchronous environment. Instead of simply fixing existing approaches (e.g., CURIOUS), we first take the chance to address the following question:

How well do existing systems leverage parallelism to handle concurrent asynchronous requests?

Indeed, existing systems have some undesirable features. CURIOUS relies on data partitioning, and accesses to the same *partition* are sequentialized. In contrast, here, we would like to develop a system which is "natively" concurrent – we would like our system to achieve high throughput even when using a single partition. ObliviStore achieves higher concurrency on individual partitions, yet, as pointed out in [4], the system relies on a fairly complex background shuffling process which is responsible for writing data back to the server and which significantly affects performance of the system.

TaoStore: Motivated by the above concerns, we develop and evaluate TaoStore, a fully-concurrent provably secure multi-user oblivious data store. TaoStore departs from the traditional partition-based SSS approach [37] used in current systems. Instead, it relies on a *tree based* ORAM scheme aimed at fully concurrent data access. Tree-based ORAMs organize server storage as a tree, and server access is in form of retrieving or overwriting data contained in a path from the root to some leaf. Our new scheme features a novel approach to manage *multiple* paths fetched concurrently from the server. In particular, the write back of updated path contents to the server occurs in an entirely *non-blocking* way, i.e., new paths *overlapping* with paths being written back can still be retrieved and updated *while* the write back operation is under way.

TaoStore is substantially simpler than ObliviStore and enables better concurrency than CURIOUS. We can in particular dispense with running the expensive background shuffling process from the former, and different from the latter, operations can be executed concurrently even on individual partitions.

Security and correctness: We prove the ORAM scheme underlying TaoStore secure using our new security framework, which guarantees security against adversaries which can schedule both operations and network messages. In particular, a key contribution of our construction is the introduction of a *sequencer* module aimed at preventing our attacks affecting other systems. Correctness (i.e., atomic semantics) remains guaranteed, regardless of the scheduling of messages sent over the network, which is asynchronous and can even be in total adversarial control. Our concurrency handling calls for a rigorous proof of correctness, which was not necessary in previous systems due to simpler approaches to accessing shared objects.

Evaluation: We present two different evaluations of TaoStore: (1) A local evaluation (with the same experimental setup as in [36]) to compare it with ObliviStore, and (2) A cloud-based evaluation (using Amazon EC2) to test our system in real-world connectivity scenarios. The first evaluation shows for example that TaoStore can deliver up to 57% more throughput with 44% less response time compared to ObliviStore. Our cloud-based evaluations show that while TaoStore's throughput is inherently limited by bandwidth constraints, this remains its main limitation – our non-blocking write-back mechanism indeed allows TaoStore's performance scale very well with increasing concurrency and decreasing memory availability at the proxy. That is, the frequency of write backs does not substantially slow down the system.

We emphasize that we *do not* implement recent bandwidth-reducing techniques using server-side computation [33], [14], [29] – we *explicitly* target usage on a simple storage server which only allows for read-write access and no computation (except for basic time-stamping), and these newer schemes – while extremely promising – are not relevant for our setting.

Partitioning: Previous works use data partitioning in a fundamental way. In particular, CURIOUS [4] relies on data partitioning to ensure concurrency (access to the same partition are sequentialized). TaoStore does not rely on partitioning – indeed, the performance of our system is competitive even without it – yet there are scenarios where partitioning is desirable, as it can help overcome storage, bandwidth, and I/O limitations. If desired, our tree-based approach enables partitioning as a simple add-in – one breaks up the tree into a forest of sub-trees, maintaining the tree-top in the proxy.

C. Overview of TaoStore

Developing an ORAM scheme for a concurrent setting is indeed far from obvious. To see why this is the case, we first review the main ideas behind tree-based ORAM schemes, such as Path ORAM by Stefanov et al. [38].

These schemes have their storage space organized as a tree, with each node containing a certain number of (encrypted) blocks. A single client keeps a position map mapping each (real) block address to a path from the root to a leaf in the tree, together with some local memory containing a (usually small) number of overflowing blocks, called the *stash*. To achieve correctness, the ORAM client maintains a *block-path invariant* ensuring that at each point in time, a block is either on its assigned path *or* in the stash. Under this invariant, processing each access (either read or write) for a block involves three operations—*read-path, flushing and write-back*. First, the ORAM client fetches the path P assigned to the block, and uses it together with the stash to answer the request. To maintain obliviousness, the block is immediately assigned to a new random path in the position map, so that a future access for the same block would fetch an independent random path (hiding repetition in accesses). Next, the contents of the path P and stash are re-arranged so that every block ends up at the lowest possible node on P and also on its assigned path; only blocks that do not fit remain in the stash. This re-arrangement is referred to as *flushing* and is crucial for ensuring that the stash never "overflows". Finally, a re-encrypted version of P is written back to the server, keeping the server up-to-date.

How do we make Path ORAM concurrent and asynchronous, while retaining both security and correctness? Even after a first glance, several issues immediately arise. First off, multiple paths may need to be retrieved simultaneously, as one request may be made while a path is being retrieved from the server – however, what if the requests are for the same item? Second, every path needs to be written back to the server, but what if just after the contents of a path have been sent back to the server, one of the items contained in this path needs to be updated? Finally, if the attacker can observe when the clients receive responses to their requests, how does one ensure that the timing of these responses are oblivious? All of this must be considered in a truly asynchronous setting, where we do not want to make any timing assumptions on the communication between the proxy and the server.

Our ORAM scheme – TaORAM – resembles Path ORAM, but allows multiple paths to be retrieved *concurrently*, without waiting for on-going flush and write-back operations to complete. All operations are done asynchronously:

- At the arrival of a request for a certain block, the appropriate read-path request is sent immediately to the server.
- Upon the retrieval of a path from the server, the appropriate read/write requests are answered, and the path is flushed and then inserted into a local *subtree* data structure.
- Immediately after flushing a certain number k of paths, their re-encrypted contents are written back to the server (and appropriate nodes deleted from the local subtree).

Here, we highlight the fundamentals of our approach, and how we address the challenges outlined above; see Section IV for more details.

Consider obliviousness: Path ORAM crucially relies on the fact that a block is assigned to a fresh new random path after each access to hide future accesses to the same block. However, in TaORAM, a request for a block is processed immediately, without waiting for other concurrent accesses to the same block to properly complete and "refresh" the assigned path. If handled naively, this would lead to fetching the same path multiple times, leaking repetition. TaORAM resolves this issue by keeping track of all concurrent requests for the same block (via a data structure called request map) so that at each point, only one request triggers reading the actual assigned path, whereas all others trigger fake reads for a random path.

Correctness is potentially jeopardized when there are multiple on-going read-path and write-back operations to the server. The most prominent issue is that before all write-back operations complete, the contents at the server are potentially out-of-date; hence answering requests using paths read from the server could be incorrect. To overcome this, TaORAM keeps a so-called *fresh-subtree* invariant: The contents on the paths in the local subtree and stash are always up-to-date, while the server contains the most up-to-date content for the remaining blocks. Moreover, every path retrieved from the server is first "*synched up*" with the local subtree, and only then used for finding the requested blocks, which is now guaranteed to be correct by the fresh-subtree invariant. Several technical challenges need to be addressed to maintain the invariant, as the local subtree and the server are constantly concurrently updated, and read-path and write-back operations are completely asynchronous.

The stash size analysis of Path ORAM breaks down when operations are asynchronous. Nevertheless, we show that the stash size of TaORAM running with a sequence of requests is the same as that of Path ORAM running with a different but related sequence of requests, which is permuted from the actual sequence according to the timing of flushing.

D. Further background and related works

It is impossible to cover the huge body of previous works on ORAM, and its applications. We have already discussed works implementing multi-client systems and in particular ObliviStore and PrivateFS – here, we give a short overview of other works.

a) Hierarchical ORAMs: Hierarchical ORAMs were first proposed by Goldreich and Ostrovsky [17] (referred to as the GO-ORAM henceforth), to store N elements. Hierarchical ORAMs organize the memory in $\log N$ many levels, consisting of increasingly many 2^i buckets. At any time point, each logical block is assigned to one random bucket per level, and stored in exactly one of them. Hierarchical ORAMs require a regular shuffling operation to deal with overflowing levels after oblivious re-insertion of items into the hierarchical data structure. Subsequent hierarchical ORAMs improve different aspects of GO-ORAM, such as reduced overhead [32], [25], [22], [20], [23], [20], faster shuffling [19], [20], [25], [43], and de-amortizing shuffling [31], [5], [21], [25], [44].

Tree ORAMs: Tree ORAMs have been proposed relatively recently, first by Shi et al. [35] and then soon extended in a number of works [38], [16], [11], [10]. The current state-of-the-art construction is Path ORAM [38] which was briefly reviewed above and will be reviewed in detail below. Other tree ORAMs share the same overall structure but differ in important details, for instance, the absence of stash in [35], [11], [16], varying heights and degrees of the tree in [16], [10], applying flushing on randomly chosen paths in [11], [10], or on paths in a fixed deterministic order [16], [33], reducing the frequency of flushing and changing the tree bucket structure [33], varying the size of the blocks [38], [40], and achieving constant communication size by moving computation to the server [14], [29].

Recent practical constructions: In the past several years, many practical ORAM schemes have been constructed and implemented for real-world applications, like secure (co-)processor prototypes [15], [26], [27], [34] and secure cloud storage systems [5], [39], [26], [44], [37], [36], [13]. While classical ORAM schemes with small client memory apply directly to the former setting, in cloud applications where a client wishes to outsource the storage of a large dataset to a remote server and later access it in an oblivious way, the client typically has more storage space, capable of storing $O(\sqrt{N})$ blocks or even some per-block meta-data of total size $O(N \log N)$. The availability of large client storage enables significantly reducing the computation overhead of ORAM to $O(\log N)$ [20], [23], [42], [41], [44], [37], [36], and furthermore, reduces the number of client-server interactions per access to $O(1)$ (instead of $O(\log N)$).

Other works on multi-client ORAM: A problem superficially related to ours (but technically different), is that of Oblivious Parallel RAM (OPRAM), recently introduced by Boyle, Chung, and Pass [6]. Even though Path ORAM-like OPRAM schemes have also been proposed [8], OPRAM clients coordinate their access to the server *without* a proxy. To achieve this, they can communicate *synchronously* with each other. The resulting schemes are however fairly unpractical.

A recent work by Maffei et al. [28] also considers ORAM in conjunction with multi-user access, developing a new primitive called *Group ORAM*. Their work considers a scenario where a data owner enforces access-control restrictions on data, whereas we consider a common address space which can be accessed by a group of mutually-trusting users. The efficiency of their solution compares to that of single-client, sequential, ORAM schemes (like Path ORAM), and they do not address efficient, high-throughput, concurrent access, which is the focus of our work.

II. Asynchronous ORAM Schemes: Definitions and Attacks

This section addresses the security of ORAM schemes in asynchronous settings. We give both a formal security model, and attacks against existing implementations.

A. Security Model

Traditional ORAM security definitions consider synchronous and non-concurrent (i.e., sequential) systems. Here, we introduce the new notion of *adaptive asynchronous obliviousness*, or aaob-security, for short. The attacker schedules read/write operation requests (which are possibly concurrent) at any point in time, and also controls the scheduling of messages. Moreover, the attacker learns *when* requests are answered by the ORAM client (i.e., the client returns an output), which as we see below, is very crucial information difficult to hide in practice. Note that the definition of [36] (which is also used in [4]) *does* consider asychronicity, but it is inherently *non-adaptive* and, even more importantly, does not reveal response times.

We give an informal (yet self-contained) overview of the definition – further formal details are deferred to Appendix A. We stress that we do not differentiate, at the formal level, between multi- and single-client scenarios – an ORAM scheme is what is run by the proxy in our application scenario, but we think more generally this of it as a single "client" answering asynchronous requests. Whether these come from multiple parties or not is orthogonal to our treatment.

b) ORAM Schemes: We think of an asynchronous ORAM scheme as a pair $\mathsf{ORAM} = (\mathsf{Encode}, \mathsf{OClient})$, where Encode takes an initial data set D of N items with a certain block size B, and produces an encrypted version $\hat{D}$ to initialize an *untrusted storage* sever SS, together with a corresponding secret key K. In particular, SS gives basic read/write access to a client accessing it, together with timestamping, i.e., writing a new item in some location on SS overwrites the current item only if the timestamp of the new item is larger. $\mathsf{OClient}$ is the actual (stateful) client algorithm which is given K, and can be invoked at any time with requests for read/write operations, and eventually answers these requests, after interacting with SS. Concretely, $\mathsf{OClient}$ processes ***read requests*** for a certain block address $\mathsf{bid} \in [N]$ to retrieve the value stored in this block, and ***write requests*** to overwrite the value of a certain block bid (and possibly retrieve the old value). These requests are denoted as $(\mathsf{op}, \mathsf{bid}, v)$ where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$ and $v = \perp$ when $\mathsf{op} = \mathsf{read}$. Every such request is *terminated* at the point in time by either returning the retrieved value or (for write operations) simply an acknowledgement to the caller, and possibly the value which was overwritten.

c) Security definition: We now proceed with our definition of aaob security, which is an indistinguishability-based security notion. Given an attacker $\mathcal{A}$ and an ORAM scheme $\mathsf{ORAM} = (\mathsf{Encode}, \mathsf{OClient})$, we consider an experiment $\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$ where $\mathsf{OClient}$ accesses a storage server SS via an asynchronous link. The experiment initially samples a random challenge bit $b \overset{\$}{\leftarrow} \{0, 1\}$, and then proceeds as follows:

- The attacker $\mathcal{A}$ initially chooses two equally large data sets D_0, D_1. Then, the game runs $(\hat{D}_b, K) \overset{\$}{\leftarrow} \mathsf{Encode}(D_b)$. As a result, $\hat{D}_b$ is stored on SS, and the key K is given to OClient.
- The attacker $\mathcal{A}$ can, at any point in time, invoke OClient with a *pair* of operation requests $(\mathsf{op}_{i,0}, \mathsf{op}_{i,1})$, where both requests can be for arbitrary read/write operations. Then, operation request $\mathsf{op}_{i,b}$ is handed over to OClient. When the operation completes, the adversary $\mathcal{A}$ is notified, yet it is not told the *actual* value returned by this operation.[1]
- When processing operation requests, OClient communicates with SS over a channel whose scheduling is controlled by $\mathcal{A}$. Concretely, when ORAM sends a read or write request to SS, $\mathcal{A}$ is notified (and given the message contents), and $\mathcal{A}$ can decide to deliver this message to SS at any point in time. Similarly, $\mathcal{A}$ controls the scheduling of the messages sent back from SS to ORAM, and also learns their contents. There are no ordering constraints – $\mathcal{A}$ can deliver messages completely out of order, and even drop messages.
- Finally, when the adversary $\mathcal{A}$ is ready, it outputs a guess b' for b, and the experiment terminates. In particular, if $b = b'$, we way that the experiments outputs `true`, and otherwise it outputs `false`.

We define the aaob-*advantage* of the adversary $\mathcal{A}$ against ORAM as

$$\mathsf{Adv}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A}) \Rightarrow \mathtt{true}\right] - 1 \, .$$

We say that ORAM is aaob-*secure* (or simply, secure) if $\mathsf{Adv}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$ is negligible for all polynomial-time adversaries $\mathcal{A}$ (in some understood security parameter λ).

d) Remarks: One key point of our definition is that the adversary learns the response times – this was not the case in [36]. This information is crucial, and in particular it is very hard to argue an adversary has no access to it. Not only in our deployment scenario this information is visible by a potential network intruder (the actual ORAM client is run by a proxy with network connectivity to its users), but also ORAM users will most likely have different behaviors triggered by these responses.

We also note that (out of formal necessity) we do not leak the *contents* of operation responses, and only their timing. Otherwise, $\mathcal{A}$ can easily recover the challenge bit b. In the full version, we discuss stronger simulation-based security notions allowing this information to be revealed.

e) Correctness: The above discussion did not address the issue of correctness of the scheme, which is quite subtle given the concurrent nature of the system. Following the classical literature on distributed systems, Appendix C defines *atomic semantics* for an asynchronous ORAM scheme as our target correctness notion. This in particular means that operations appear to take place atomically at some point between their invocation and their response.

[1]This restriction is necessary, for otherwise an adversary $\mathcal{A}$ could easily guess the value of b.

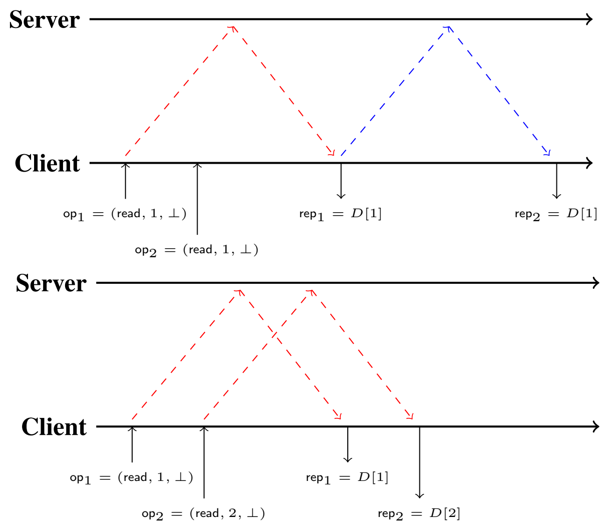

Server
Client
op_1 = (read, 1, ⊥)
op_2 = (read, 1, ⊥)
rep_1 = $D[1]$
rep_2 = $D[1]$
Server
Client
op_1 = (read, 1, ⊥)
op_2 = (read, 2, ⊥)
rep_1 = $D[1]$
rep_2 = $D[2]$

Fig. 1: **Attack against ObliviStore.** Comparison of event timing for repeated access (above) and distinct accesses (below). Here, we assume constant delays in delivering messages.

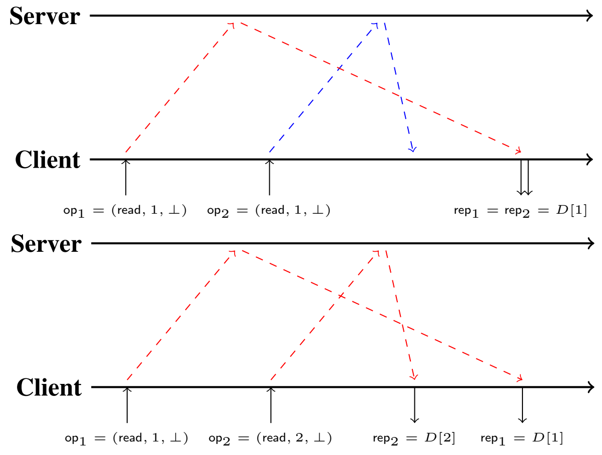

Server
Client
op_1 = (read, 1, ⊥)
op_2 = (read, 1, ⊥)
rep_1 = rep_2 = $D[1]$
Server
Client
op_1 = (read, 1, ⊥)
op_2 = (read, 2, ⊥)
rep_2 = $D[2]$
rep_1 = $D[1]$

Fig. 2: **Attack against CURIOUS's fake-read logic:** The upper figure represents the timing of the communication between the client and the server when accessing the same item twice, and the second access is a "fake read" (in blue). The figure below represents the execution when the accesses are for two distinct items (both "real reads"). The timings of the responses differ, as in the above case, the client needs to wait for the actual value to arrive.

B. Attacks

We present two attacks – one against ObliviStore, one against CURIOUS – breaking their aaob-security. We note that the former attack is just a re-iteration of the key idea presented in [4]. In contrast, our second attack is novel. We give a high-level explanation of the attacks, but a formalization in our framework (given an appropriate formalization of the scheme) can be obtained easily.

f) Attack against ObliviStore: An attack against ObliviStore can be derived from the weakness already observed in [4]. In particular, ObliviStore sequentializes accesses on the same item, and thus an adversary requesting the same item *twice* (e.g., issuing two subsequent requests $\mathsf{op}_{1,0} = \mathsf{op}_{2,0} = (\mathsf{read}, 1, \perp)$) will see only one request being made to the storage server, with a second request being scheduled only *after* the response to the first one returns to the client. In contrast, scheduling requests $\mathsf{op}_{1,1} = (\mathsf{read}, 1, \perp)$ and $\mathsf{op}_{2,1} = (\mathsf{read}, 2, \perp)$ for two different addresses will have the adversary see the client immediately schedule two requests to retrieve information from the server. This leads to easy distinguishing. Figure 1 gives two diagrams presenting the two situations in detail.

We note two things. First off, this attack breaks ObliviStore even in the model in which it was claimed to be secure, as response times are not needed to distinguish between the repeated-access scenario. Also, the attack does not require the network to be asynchronous – only the ability to schedule overlapping operations. Second, if response times can be measured, then the attack is very easy to mount: An independent experimental validation (with the ObliviStore implementation provided to us) shows that repeatedly accessing the same item over and over leads to a performance degradation of up to 50% compared to accessing well-spread loads.

g) Attack against CURIOUS: The overcome this, [4] suggested an alternative approach based on the idea that a concurrent operation on the same item should trigger a "fake read". We show that this idea, by itself, is not sufficient to achieve aaob-security. We note that our attack does not contradict security claims in [4], since the model of [36] is used, which does not leak the timing of responses. (As argued above, we believe that it is extremely hard to hide these timings in actual deployment.)

To start with, recall that when two concurrent requests for the same item are made in CURIOUS (think of these as read requests for simplicity), the first request results in the actual "real read" access to the server fetching the item, whereas the second results in a fake access to the storage server SS (a so-called "fake read") to hide the repeated access. This "fake read" looks like an access to an unrelated, independent item (the details are irrelevant).

The key issue – ultimately allowing us to distinguish – concerns the *timings* of the responses given by the ORAM client. When the fake read operation terminates (i.e., the corresponding data is received by the ORAM client from the server), the client always returns the item fetched in the real read *if it is available*. If the item is not available, then it needs to wait for the real read to terminate. Note that in the asynchronous setting, the latter situation *can* occur – we have no guarantee whatsoever that the real read terminates before

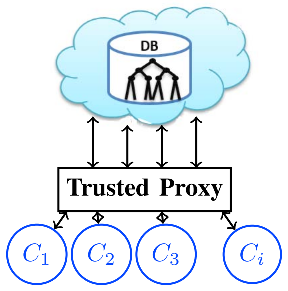


Fig. 3: Deployment model of TaoStore

the fake read.[2] This is in contrast to the case where the reads are for two distinct items (and hence both "real"), and the second request can be answered right away even if the client has not received the data from the server associated with the second request.

This gives the attacker a simple mean to break aaob security, and distinguish the $b = 0$ from the $b = 1$ case, by simply scheduling two pairs of operations $(\mathsf{op}_{1,0}, \mathsf{op}_{1,1}), (\mathsf{op}_{2,0}, \mathsf{op}_{2,1})$, where $\mathsf{op}_{1,0}$ and $\mathsf{op}_{2,0}$ are two read requests for the same item, whereas $\mathsf{op}_{1,1}$ and $\mathsf{op}_{2,1}$ are read requests for distinct items. Concretely, the adversary $\mathcal{A}$ first issues the request pair $(\mathsf{op}_{1,0}, \mathsf{op}_{1,1})$, delays the messages sent by OClient right after the first operation pair is processed, schedules the second request pair $(\mathsf{op}_{2,0}, \mathsf{op}_{2,1})$, and delivers the associated messages to SS, and its replies back to OClient immediately. If this results in an answer to the second operation being triggered immediately, the attacker guesses $b = 1$, otherwise it guesses $b = 0$. The outcome of the attack is depicted in Figure 2.

h) Remarks: We note that to prevent the same attack affecting CURIOUS, our system TaoStore will introduce the notion of an *operation sequencer*, a module catching out-of-order early replies from the ORAM client back to the caller, for instance by ensuring that in our attack scenario from above, *also* in the setting with two real reads, the final response to the second real read will not be sent before the response to the first real read. In other words, we will not happen to modify the fake-read logic. Rather, we make sure that real reads have response timings consistent with the behavior one would observe if some of these are fake.

III. Overview of TaoStore

This section provides a high-level overview of TaoStore and its goals, including the deployment scenario and architecture of our system.

High-level goal: The goal of TaoStore is to allow multiple clients (or users) to securely and obliviously access their shared data on an untrusted storage server (a "public cloud"). Informally, the security guarantee is that the contents of the

[2]CURIOUS in fact envisions the fake read going with high probability to a partition different than the real read – this partition may even be on a different machine, and thus out-of-order responses are quite likely.

shared data and of the accesses from the multiple clients are kept hidden against any honest-but-curious entity[3] observing traffic to and from the server and being able to schedule messages. This is formalized via the notion of aaob security introduced above.

Concretely, users issue *read requests* for a certain block address bid to retrieve the value stored in this block, and *write requests* to overwrite the value of a certain block bid (and possibly retrieve the old value). These requests are denoted as $(\mathsf{type}, \mathsf{bid}, v)$ where $\mathsf{type} \in \{\mathsf{read}, \mathsf{write}\}$ and $v = \perp$ when $\mathsf{type} = \mathsf{read}$. The block address bid belongs to some logical address space $\{1, \ldots, N\}$, and blocks have some fixed size B. (In our system, $B = 4$ KB.) Every such request is *invoked* at some point in time by a client process, and *terminates* at the point in time by either returning the retrieved value or (for write operations) simply an acknowledgement to the caller.

System architecture: As in previous works [36], [4], TaoStore relies on a *trusted proxy*, who acts as a middle layer between users and the untrusted storage. (See Figure 3 for an illustration of the architecture.) The proxy coordinates accesses from multiple users to the untrusted storage, which it makes oblivious, and stores locally secret key material used to encrypt and decrypt the data stored in the cloud. We also assume that the communication between users and the proxy is protected by end-to-end encryption. This is often referred to as the "hybrid cloud" model [36].

TaoStore's proxy will effectively run the Oblivious RAM scheme, TaORAM (briefly discussed above in the introduction and presented below in Section IV), which is particularly well suited at processing requests in a highly concurrent way, as opposed to traditional ORAM schemes which would force request processing to be entirely sequential.[4] We assume that network communication, most importantly between the proxy and the untrusted storage, is completely *asynchronous*. Furthermore, in contrast to classical applications, the ORAM scheme here can effectively use large memory on the proxy, even up to $N \log N$ (e.g., to store a full position map). (Large proxy memory was also exploited in ObliviStore already.)

IV. Our Asynchronous ORAM

In this section, we present the asynchronous ORAM scheme underlying TaoStore – which we refer to as TaORAM. In particular, TaORAM is run by the trusted proxy, which acts as the "single client" interacting with the storage server, handling queries concurrently. Therefore, in the following, we refer to the entity running the ORAM algorithm (the trusted proxy here) as the *ORAM client*.

TaORAM is based on the non-recursive version of Path ORAM, but processes client requests *concurrently and asynchronously*. We focus on the non-recursive version, since in our deployment model the trusted proxy has reasonably large memory, able to hold some meta-data for each data block. (The same recursive technique as in Path ORAM can be applied to reduce the memory overhead if needed.) Below, we first briefly review Path ORAM, and then describe TaORAM.

A. A Review of Path ORAM

To implement a (logical) storage space for N *data blocks* (stored in encrypted form) the basic Path ORAM scheme organizes the storage space virtually as a complete binary tree with at least N leaves, where each node of the tree is a small storage bucket that fits $Z = 4$ data blocks. To hide the logical access pattern, each data block is assigned to a random path pid from the root to the leaf (so we can equivalently think of pid as being the identifier of a leaf, or of such a path) and stored at some node on this path; the assignment is "refreshed" after each access for this block (either for a read or for a write operation) to a new random path pid' to hide future accesses to the same block. The ORAM client keeps track of the current assignment of paths to blocks using a *position map*, pos.map, of size $O(N \log N)$ bits,[5] overflowing blocks (see below) in an additional data structure, called the *stash*, and denoted stash, of fixed a-priori bounded size (the size can be set to some function of the order $\omega(\log N)$, even only slightly super-logarithmic).

For each client request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ with $\mathsf{type}_i = \mathsf{read}/\mathsf{write}$, Path ORAM performs the following operations:

1) **Request Processing (Read-Path):** Upon receiving the request, Path ORAM sends a read request to the server for the path $\mathsf{pid} = \mathsf{pos.map}[\mathsf{bid}]$ assigned to block bid. When the path is retrieved, it decrypts the path and finds block bid on the path or in stash, and either returns its value if $\mathsf{type}_i = \mathsf{read}$, or updates it to v_i if $\mathsf{type}_i = \mathsf{write}$. Path ORAM then assigns block bid to a new random path pid' and updates $\mathsf{pos.map}[\mathsf{bid}] = \mathsf{pid}'$ accordingly.
2) **Flushing:** In a second phase, it iterates over each block bid on the path pid or in the stash, and inserts it into the lowest non-full node (i.e., containing less than Z nodes) on pid that intersects with its assigned path $\mathsf{pos.map}[\mathsf{bid}]$. If no such node is found, the block is placed into the stash.
3) **Writing-back:** Then Path ORAM encrypts the path with fresh randomness, and writes path pid back to the server.

Initializing the remote storage. To initialize the contents of the remote storage server, the ORAM client can simply run the ORAM algorithm locally, inserting elements one by one. The resulting storage tree can be safely sent to the server to store and accessed later. Since this approach can be applied universally to any ORAM scheme, we omit a discussion on encoding the initial data set below.

B. TaORAM

TaORAM internally runs two modules, the *Processor* and the *Sequencer*. (See Figure 4 for an illustration.) The Processor

[3]While not addressed in this paper, enhancing security to an actively malicious server can be achieved via fairly standard techniques.

[4]The number of clients is irrelevant for our system, as all clients are allowed to access the same data and each client can issue multiple queries concurrently, and thus effectively an arbitrary number of clients can be seen as one single client accessing the proxy without loss of generality.

[5]The full Path ORAM scheme recursively outsources the position map to the server to reduce the ORAM client's local storage to $\mathrm{poly}\log(N)$.

interacts with the server, prepares answers to all logical requests, and returns answers to the Sequencer. The Sequencer merely forwards logical requests to the Processor, and when receiving the answers, enforces that they are returned in the same order as the requests arrive, as we explain in more detail below.

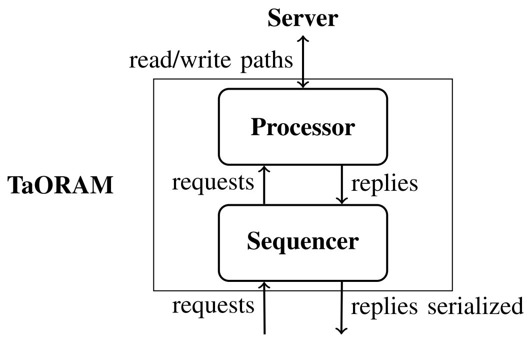


Fig. 4: TaORAM Structure

We present TaORAM in steps. Step 1-3 describe the design of the Processor, each step enabling a higher degree of concurrency. In this description, when obliviousness is concerned, it is convenient to focus only on the communication between the Processor and the server. Then, in Step 4, we show how to prevent additional information leakage through the timing of replies, and in particular explain the functionality of the Sequencer. A complete pseudocode description of TaORAM is provided in Figure 5.

1) Step 1 – Partially Concurrent Requests: For any $k \geq 1$, Path ORAM can naturally be adapted to support partial "k-way" concurrent processing of logical requests when the k logical requests are *non-repetitive* (i.e., accessing distinct blocks).[6] In this case, the Processor implement a variant of Path ORAM to first (1') simultaneously fetch k paths from the server to find the requested blocks, and store all paths in local memory, forming a subtree we refer to as subtree; after assigning these k blocks to k new random paths, (2') it flushes along the subtree, and (3') writes back the entire subtree to the server. Note that since the server is not updated during step (1'), the read-path requests for the k logical requests can be issued concurrently and asynchronously, without further coordination. Furthermore, when logical requests are for distinct blocks, the k paths fetched in step (1') are independent and random, and this ensures obliviousness.

However, when there are repetitive logical requests, obliviousness no longer holds. This is because multiple accesses to the same block cause the Processor to fetch the same path multiple times, leaking the existence of repetition. To solve this issue, TaORAM maintains a **request map**, denoted as request.map, that maps each block bid to a queue, request.map[bid], of (unanswered) logical requests for this block. To avoid leaking repetitions, only the *first* logical request in the queue triggers reading the actual assigned path—termed a "real read", whereas all following requests trigger reading a random path—termed a "fake read". Later, when the assigned path is retrieved, responses to all requests in request.map[bid] are created in sequence to ensure logical consistency. (See Step 2 in algorithm READ-PATH and Step 3 in algorithm ANSWER-REQUEST in Figure 5.)

[6]A similar observation was made for hierarchical ORAMs in the design of PrivateFS [44], which supports partial concurrent processing of requests from multiple clients.

2) Step 2 – Fully Concurrent Request Processing: In the above scheme, flush and write-back operations (i.e., Step 2' and 3') implicitly "block" the processing new requests, imposing an undesirable slow down. In the following, we enhance the Processor to enable *fully* concurrent processing: Each incoming request is immediately inserted into the request map and the appropriate path is fetched from the server, *even if flushing and writing back of previously retrieved paths are in progress.*

Such modification brings a number of challenges for ensuring correctness. For example, before a write-back operation is completed, part of the contents on the server are potentially stale, and hence reading a path from the server at the same time may lead to an incorrect answer to some logical request. To ensure correctness, TaORAM will maintain the following,

> **Fresh-Subtree Invariant**: *The blocks in the local* subtree *and* stash *are always up-to-date, whereas the tree at the server contains the most up-to-date contents for the remaining blocks.*

The invariant is strongly coupled with our subtree **synching procedure**: Whenever the Processor retrieves a path from the server, it discards the part that intersects with the local subtree, and only inserts the rest of the nodes into subtree. Under the fresh-subtree invariant, after "synching", the path in subtree is guaranteed to be up-to-date, and can safely be used to answer logical requests. (See Step 1 of algorithm ANSWER-REQUEST in Figure 5.)

Maintaining the invariant is, however, subtle, and one of the core technical challenges in our algorithm. If nodes in subtree were never deleted, the invariant would be trivially maintained, as all updates are first performed on subtree. But, this eventually leads to a huge subtree. Therefore, whenever the server confirms that some k paths has been written back, the Processor deletes some nodes from subtree.

Unfortunately, naively deleting the entire k paths would violate the fresh-subtree invariant. This is because between the time t_1 when the write-back operation starts and t_2 when it completes (receiving confirmation from the server), the subtree is potentially updated. Hence, at t_2, the Processor must keep all nodes updated after t_1, or else new contents would be lost. Another issue is that between t_1 and t_2, new logical requests may trigger reading a path pid from the server; to ensure that when the path is retrieved (after t_2), it can be correctly "synched" with subtree, the Processor must keep all nodes on path pid (for the content retrieved from the server may be stale since the path is requested before t_2).

In summary, the Processor must not delete any nodes that have been more recently (than t_1) updated or requested.

```
Module Sequencer:
Global Data: A sequencer.queue and a sequencer.map.
Sequencer reacts to the following events:
  • Upon receiving request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$, do:
      – Create entry $\mathsf{sequencer.map}[(\mathsf{type}_i, \mathsf{bid}_i, v_i)] \leftarrow \perp$.
      – Push request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ into sequencer.queue.
      – Send request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ to Processor.
  • Upon receiving response $w_i$ for request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ from
    Processor, set $\mathsf{sequencer.map}[(\mathsf{type}_i, \mathsf{bid}_i, v_i)] \leftarrow w_i$.
  • Run on a separate thread the Serialization Procedure that keeps
    doing the following:
      – When sequencer.queue is non-empty, pop a request
        $(\mathsf{type}, \mathsf{bid}, v)$ from sequencer.queue.
      – Wait until entry $\mathsf{sequencer.map}[(\mathsf{type}, \mathsf{bid}, v)]$ is updated
        to a value $w \neq \perp$.
      – Return $w$ as a response to request $(\mathsf{type}, \mathsf{bid}, v)$, and
        remove entry $\mathsf{sequencer.map}[(\mathsf{type}, \mathsf{bid}, v)]$.

Module Processor:
Global Data: A secret (encryption) key key, a stash, a request.map,
a response.map, a PathReqMultiSet, a subtree, a counter #paths
and a write.queue.
Processor reacts to the following events:
  • Upon receiving a logical request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ from Se-
    quencer, start a new thread doing the following and then
    terminate.
      – (pid, $P$, fake.read) ← READ-PATH($\mathsf{type}_i, \mathsf{bid}_i, v_i$);
      – Lock subtree;
      – ANSWER-REQUEST($\mathsf{type}_i$, $\mathsf{bid}_i$, $v_i$, pid, $P$, fake.read);
      – FLUSH(pid);
      – Unlock subtree;
  • Whenever #paths turns a multiple of $k$, $c \cdot k$, start a new
    thread running WRITE-BACK($c$);

READ-PATH($\mathsf{type}_i, \mathsf{bid}_i, v_i$):
  1) Create entry $\mathsf{response.map}[(\mathsf{type}_i, \mathsf{bid}_i, v_i)] \leftarrow (\mathsf{false}, \perp)$.
  2) Insert $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ into queue $\mathsf{request.map}[\mathsf{bid_i}]$.
       • If the queue was previously empty, set fake.read ← 0
         and pid ← $\mathsf{pos.map}[\mathsf{bid}_i]$;
       • Else, set fake.read ← 1, and sample pid $\overset{\$}{\leftarrow} \{0,1\}^D$.
  3) Read-path pid from server and insert pid to
     PathReqMultiSet. Wait for response.
  4) Upon waking up with the server response, remove (one
     occurrence of) pid from PathReqMultiSet.
  5) Decrypt the response with key to obtain the content of path
     pid, denoted as $P$, and return (pid, $P$, fake.read).

ANSWER-REQUEST($\mathsf{type}_i$, $\mathsf{bid}_i$, $v_i$, pid, $P$, fake.read):
  1) Syncing procedure: Insert every node $w$ on path $P$ that
     is currently not in subtree into subtree.
  2) Update entry $\mathsf{response.map}[(\mathsf{type}_i, \mathsf{bid}_i, v_i)]$ from
     $(b, x)$ to $(\mathsf{true}, x)$. If $x \neq \perp$, reply value $x$ for the
     request $(\mathsf{type}_i, \mathsf{bid}_i, v_i)$ to Sequencer, and delete the
     entry.
  3) If fake.read $= 0$, find block $\mathsf{bid}_i$ in subtree, and create
     responses to requests in queue $\mathsf{request.map}[\mathsf{bid_i}]$ as
     follows:
       • Pop a request $(\mathsf{type}, \mathsf{bid}_i, v)$ from the queue.
       • Let $w$ be the current value of block $\mathsf{bid}_i$.
       • If type = write, set the value of $\mathsf{bid}_i$ to $v$.
       • If entry $\mathsf{response.map}[(\mathsf{type}, \mathsf{bid}_i, v)] = (\mathsf{true}, \perp)$,
         reply value $w$ for the request $(\mathsf{type}, \mathsf{bid}_i, v)$ to
         Sequencer, and delete the entry.
       • Else, if $\mathsf{response.map}[(\mathsf{type}, \mathsf{bid}_i, v)] = (\mathsf{false}, \perp)$,
         set the entry to $(\mathsf{false}, w)$.
     Repeat the above steps until $\mathsf{request.map}[\mathsf{bid_i}]$ is empty.
  4) If fake.read $= 0$, assign block $\mathsf{bid}_i$ a new random path
     $\mathsf{pos.map}[\mathsf{bid}_i] \overset{\$}{\leftarrow} \{0,1\}^D$.

FLUSH(pid):
  1) For every block bid′ on path pid in subtree and stash,
     do:
       • Push block bid′ to the lowest node in the intersec-
         tion of path pid and pos.map[bid′] that has less
         than $Z$ blocks in it. If no such node exists, keep
         block bid′ in stash.
  2) Increment #paths and push pid into queue write.queue.
  3) For every node that has been updated, add (local)
     timestamp $t =$ #paths.

WRITE-BACK($c$):
  1) Pop out $k$ paths $\mathsf{pid}_1, \cdots \mathsf{pid}_k$ from write.queue.
  2) Copy these $k$ paths in subtree to a temporary space $S$.
  3) Encrypt paths in $S$ using secret key key.
  4) Write-back the encrypted paths in $S$ to the server with
     (server) timestamp $c$. Wait for response.
  5) Upon waking up with write confirmation, delete nodes
     in subtree that are on paths $\mathsf{pid}_1, \cdots \mathsf{pid}_k$, with (local)
     timestamp smaller than or equal to $c \cdot k$, and are not on
     any path in PathReqMultiSet.
```

Fig. 5: Pseudocode description of TaORAM.

To ensure the former, we timestamp every node in subtree (locally) to record when it is last updated. (See Step 3 of Algorithm FLUSH in Figure 5, and note that this timestamp is different from the version number used as a server timestamp.) To ensure the latter, the Processor maintains a multi-set PathReqMultiSet that tracks the set of paths requested but not yet returned.[7] (See Step 3 and 4 of algorithm READ-PATH and Step 6 of WRITE-BACK in Figure 5.)

3) Step 3 – Non-Blocking Flushing: So far, though requests are concurrently processed at their arrival, the flush and write-back operations are still done sequentially, in the same order their corresponding logical requests arrive (in batches of k). We further remove this synchronization.

First, we decouple the order in which paths are flushed from the order in which logical requests arrive: As soon as a path is retrieved ("synched" with the subtree, and used for answering client request), the Processor flushes the path immediately, even if the paths for some previous requests have not yet been returned (remember that they could well be late due to the asynchronous nature of the network). Furthermore, we make write-back operations asynchronous: As soon as k new paths are inserted into subtree and flushed, the Processor writes-

[7]We remark that PathReqMultiSet must be necessarily a multi-set, as the same path may be requested more than once.

back these k paths to the server, irrespective of the status of any other operations (e.g., some previous write-back requests may still be pending)— therefore, in the rest of the paper, we call k the *write-back threshold*. In summary, flush and write-back operations are performed as soon as they are *ready* to be performed. (See the pseudocode of Module Processor.)

This brings two challenges. First, since paths may be flushed in an order different from that they were requested, it is no longer clear whether the stash size is bounded (at least the analysis of Path ORAM does not directly apply as a black box). We show that this is indeed the case, and provide the proof below.

Lemma 1. *The stash size of TaORAM is bounded by any function $R(N) = \omega(\log N)$ (e.g. $R(N) = (\log N) \cdot (\log\log\log N)$), except with negligible probability in N.*[8]

The second challenge is ensuring server consistency when multiple write-back operations end up being concurrent. In an asynchronous network, these requests may arrive at the server out-of-order, causing the server to be updated incorrectly. To address this problem, we mark each node stored at the server, as well as each write-back request, with a version number (or "server timestamp"), and the server can only overwrite a node if the write-back request is of a newer version. (See Step 4 of WRITE-BACK; we omit the server algorithm due to lack of space.)

Proof of Lemma 1: We only give a proof sketch. A more formal proof is rather tedious and requires repeating many of the technical steps in the stash analysis of Path ORAM with little change.

We show that given any execution trace T of TaORAM with a sequence of logical requests $r_1, r_2, \cdots$, one could come up with another sequence $r'_1, r'_2, \cdots$ of the same length (modified and permuted from the original sequence based on the execution trace) which when fed to Path ORAM sequentially yields the same stash.

By design of TaORAM, whenever the Processor receives a request $r_i = (\mathsf{type}_i, \mathsf{bid}_i, v_i)$ with $\mathsf{type}_i = \mathsf{read}/\mathsf{write}$, it immediately issues a path-read request to the server, fetching either the path $\ell_i = \mathsf{pos.map}(\mathsf{bid}_i)$ assigned to block bid_i (in the case of real read), or a randomly chosen path $\ell_i \xleftarrow{\$} U$ (in the case of fake read). Furthermore, upon receiving the path ℓ_j corresponding to request r_j from the server, the Processor flushes the path immediately. The execution trace T contains the time t_j at which each path ℓ_j corresponding to request r_j is flushed. Order the time points chronologically $t_{j_1} < t_{j_2} < \cdots$. We observe that the contents of the stash are determined by the sequence of events of flushing over paths $\ell_{j_1}, \ell_{j_2}, \cdots$, where if the j_k'th request corresponds to a real read, then the block bid_{j_k} is assigned to a new path, and if the j_k'th request corresponds to a fake read, no new assignment occurs.

Suppose we execute Path ORAM with a sequence of requests $r'_1, r'_2, \cdots$ sequentially, where $r'_k = r_{j_k}$ if the j_k'th request corresponds to a real read, and otherwise r'_k is a "special request" for flushing path ℓ_{j_k} without assigning new paths to any blocks, (and suppose that the same random coins are used for assigning new paths as in execution trace T). At any point, the contents of the stash is identical to that of TaORAM with execution trace T.

It was shown in [38] that the stash size of Path ORAM when executed without "special requests" is bounded by any function $R(N) = \omega(\log N)$ with overwhelming probability. Since the "special requests" only involve flushing a path without assignment new paths (in other words, they only put blocks at lower positions on the path), the probability that the stash size exceeds $R(N)$ decreases. Therefore, the stash size of TaORAM is also bounded by $R(N)$ with overwhelming probability. ■

4) Step 4 – Response Timing and Sequencer: The above description considers only the obliviousness of the communication between the server and the Processor. Indeed, by the use of "fake reads", every read-path request to the server fetches an independent random path. Their timing, as well as that of the write-back requests, are completely determined by the timing of (the arrival of) logical requests and the schedule of asynchronous network. Hence, the Processor-server communication is oblivious of the logical requests.

Another aspect that has been neglected (on purpose) so far is the timing of replies (to logical requests). Consider the scenario where a sequence of repetitive logical requests arrives in a burst, triggering a real read (for the assigned path), followed by many fake reads (for random paths). When the real read returns, the requested block is found; but, if the Processor replies to all logical requests in one shot and an adversary observes this event, it can infer that there are likely repetitions. To eliminate this leakage, the Processor only replies to a request when the corresponding read-path request has returned, even if it is a fake read. To achieve this, the Processor uses a **response map**, denoted as $\mathsf{response.map}$, that maps each request $(\mathsf{type}, \mathsf{bid}, v)$ to a tuple $\mathsf{response.map}[(\mathsf{type}, \mathsf{bid}, v)] = (b, w)$ indicating whether this request is ready to be replied to (i.e., $b = \mathsf{true}$ if the corresponding read-path request has returned) and what the answer w is. A request is replied to only when both $b = \mathsf{true}$ and $w \neq \perp$. (See Step 2 and 3 of ANSWER-REQUEST.)

Unfortunately, a more subtle leakage of information still exists in an asynchronous network, and is exploited by our attack against CURIOUS in Section II-B. To see this, consider again the above scenario with one real-read followed by many fake reads. If in addition the real-read is indefinitely delayed due to the asynchrony of the network, the requested block is not retrieved and none of the requests can be answered (even if all fake-reads return without delay). This delay of replies again leaks information; we have explained how an adversary can use this information to violate obliviousness in Section II-B. In order to prevent this attack, TaORAM runs an additional auxiliary module, the **Sequencer**, whose sole function is enforcing that logical requests are replied to in the same order as they arrive.

[8]In fact, the statement can be made more concrete, as the probability of overflowing is roughly c^{-R} for some constant c and stash size R.

C. Client Memory Consumption

The client memory of an ORAM scheme contains both temporary data related to on-going processing of requests, and permanent data that keeps the state of the ORAM scheme. Since the latter needs to be stored even when there is no request present, it is also called the client storage. In TaORAM, the client storage consists of the position map, the stash, and the secret key key, of size respectively $O(N \log N)$, $\omega(\log N)$, and λ (the security parameter); thus,

$$\text{TaORAM Client Storage Size} = O(N \log N + \lambda) \ ,$$

which is the same as Path ORAM.

On the other hand, unlike Path ORAM and other sequential ORAM schemes, the size of temporary data in TaORAM (and other concurrent cloud storage system such as [36]) depends on the number I of concurrent "incomplete" (more details below) logical requests. The number I in turn depends on various (dynamically changing) parameters, from the rate of arrival of logical requests, to the schedule of asynchronous network, to the processing power of the server and client. Hence, we analyze the size of temporary data w.r.t. I. For TaORAM, we say that (the processing of) a logical request is incomplete, if it has not yet been answered, or updates induced by the request (due to being a write request itself and/or flushing) has not been committed to the server. For each incomplete request, TaORAM keeps temporary data of size $O(\log N)$, leading to

$$\text{TaORAM Temporary Data Size} = O(I \log N) \ .$$

In a normal execution where the rate of processing and the rate of arrival of logical requests are "balanced", since TaORAM writes-back to the server after every k paths are retrieved and flushed, the number I of incomplete requests is roughly k; hence,

$$\begin{aligned}&\text{Normal TaORAM Memory Consumption}\\ &\qquad\qquad = O(k \log N + N \log N + \lambda) \ .\end{aligned}$$

Of course, a malicious adversary can drive the number I to be very large, by simply preventing write-back operations to complete. When this is a concern, we can let the system halt whenever I reaches a certain threshold (note that I is known to the adversary, and thus this operation does not break obliviousness of the scheme).

D. Partitioning

It may be often advantageous to store our tree in a distributed fashion across multiple partitions, e.g. to prevent I/O and bandwidth bottlenecks.

TaORAM is easily amenable to partitioning, without the need of storing an additional partition table as in previous systems [36], [37], [4]. If $m = 2^i$ partitions are desired, we can simply "remove" the top i levels of the tree, storing them in TaORAM's local memory. (Note that this requires storing $O(m)$ additional data blocks locally, but this number is generally not too large.) Then, the rest of the tree can be thought as a forest of m sub-trees (the root of each sub-tree is one of the nodes at the i-th level of the original tree). One can then store each of these sub-trees on a different partition.

Note that the scheme remains unchanged – the only difference is in the data-fetch logic. The tree is now distributed across m partitions, and the TaORAM's local memory. When a path is to be fetched, one retrieves the contents of the first i levels on the path from the local memory, and the remaining levels from the appropriate partition. Every access being on a random path, the load on the partitions is uniformly distributed.

E. Security

The following theorem summarizes our security statement for TaORAM. The proof, given in Appendix B, follows from two facts: First, from our use of the sequencer module, ensuring that the i-th operation is *not* answered until all previous operations are answered. Second, from the fact that all requests retrieve random paths.

Theorem 1 (TaORAM security)**.** *Assume that the underlying encryption scheme is IND-CPA secure. Then TaORAM is* aaob-*secure.*

F. Correctness

It is a priori not clear whether the system behaves as expected, or say (for example) we may return inconsistent or outdated values for different requests. Proving correctness of the scheme, therefore, becomes a non-trivial issue in the asynchronous setting (which is in fact even *harder* than proving security). In Appendix D, we prove that TaORAM exhibits atomic semantic, i.e., completed operations appear (to an external observer) as if they took effect *atomically* at some point during their invocation and their response. (We provide formal definitions for correctness in Appendix C.)

The core of the proof lies in showing that the fresh-subtree invariant mentioned above always holds (i.e., the contents in the local storage at the proxy is the most up-to-date). Operations then take effect when a write operation writes its value into, or when a value is retrieved from the proxy's local storage.

Remark. We note that packet dropping or delays have a very isolated impact on TaORAM. Indeed, loss of some of the read-path/write-back operations will not result in stalling the system (just in slightly increased memory consumption). This is in sharp contrast to the background shuffling process of ObliviStore [36], which cannot be halted at any point as otherwise the system will stall.

V. Experiments

The experiments evaluate TaoStore in two different test environments: simulation based and real world deployment. We start by providing a detailed analysis of TaoStore's performance by deploying the untrusted server to a public cloud (AWS[1]). We then compare TaoStore with ObliviStore and Path ORAM in the hybrid cloud setting using a simulation

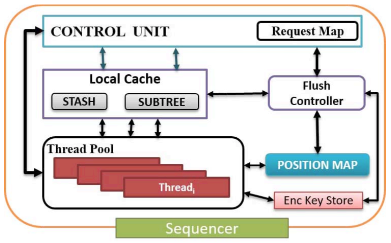


Fig. 6: Trusted Proxy Implementation

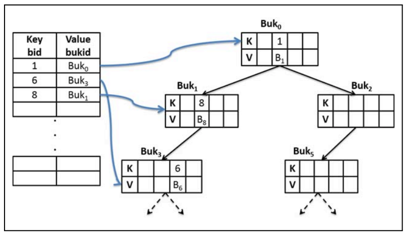


Fig. 7: Subtree Structure

based environment, which is similar to the setting in ObliviStore paper.

A. Implementation

We implemented a prototype of TaoStore in C#. We start by briefly highlighting some technical aspects of our implementation.

The trusted proxy (see Figure 6) runs an implementation of TaORAM as described in Section IV, which internally runs many *threads*, where each is a processing unit responsible for handling a client request and then returning a response to the client. The *request map* is implemented as a dynamic dictionary in the format of *(*bid*, queue)* pairs where block id, bid, is a key in the map and each value is a queue object that keeps track of the threads waiting for block bid. Additionally, the *control unit* communicates with the threads and the flush controller to maintain the state of the system. The *position map* is an array based data structure. The proxy also has a local cache with 2 components: a subtree and a stash. The subtree is implemented as a dynamic data structure that takes advantage of a dictionary and a tree structure as shown in Figure 7. For faster lookup, the dictionary component maintains the information for mapping the blocks to buckets. If a block is stored in the *subtree*, the dictionary points to the bucket in which the block is stored. The nodes themselves also use a dictionary structure to store blocks. Maintaining this two-level structure enables an $O(1)$ lookup for stored blocks. The other caching component, the *stash*, has a dictionary format of (bid, $block$). To provide data confidentiality, the data is encrypted at the bucket level using a semantically secure randomized encryption scheme, AES-128 [30] in CBC-mode, before it is outsourced to the cloud storage.

The components of the local cache are implemented in memory. When paths are fetched from the untrusted cloud storage, concurrent fetches are likely to have overlapping buckets, especially at the top levels of the tree. To avoid locking the complete subtree (which would be very costly), we apply the *read-write lock* mechanism [12] at the bucket level to control concurrent accesses to the shared buckets in the local cache.[9] If a thread wants to perform an insert, an update or a delete operation on a bucket, it has to acquire a write lock for this bucket, to which it gains exclusive access. In contrast, for read operations, it is enough for the thread to acquire a read lock, which still allows several threads to access the same bucket for reading at the same time. The stash is another shared data structure that needs to be controlled. Since it is a block level data storage, we apply read-write locks at the block level. The control unit also uses block level read-write locks to maintain concurrent operations on the request map.

Our server implementation performs I/O operations directly on the disk. TaoStore is an I/O intensive infrastructure, and for higher performance it is important to minimize the I/O overhead. Our implementation performs I/O operations at the path level, i.e., reading or writing the buckets along the path at once, rather than at the bucket level, which would require separate I/O operations for each bucket. Performing I/O at the bucket level requires more I/O scheduling and context-switch overheads; therefore TaoStore avoids it. The server responses are returned with *callbacks* which have significant performance advantages over thread pooling and scheduling.

TaoStore can cache the top levels of the tree and serve directly from memory to eliminate a significant amount of I/O overhead in the untrusted cloud storage. In our implementation, caching is done using a dictionary data structure.

In real world deployment scenario, the trusted proxy and the server communicate and exchange data over asynchronous TCP sockets.

B. Experimental Setup

The first set of experiments are conducted to analyze how TaoStore performs as an oblivious cloud storage in the real world. The trusted proxy runs on a machine on a university network with i5-2320 3 GHZ CPU, Samsung 850 PRO SSD, and 16 GB memory. The cloud storage server is deployed to an i2.4xlarge Amazon EC2 instance. The average round-trip latency from the trusted proxy to the storage server is 12 ms. The average downstream and upstream bandwidths are approximately 11 MBytes/s[10].

The second set of experiments are conducted to compare TaoStore with ObliviStore. To be comparable with ObliviStore, we use a configuration which is similar to the ObliviStore paper. The network communication between the trusted proxy and the storage server is simulated with a 50 ms latency. Although there are multiple clients and they query the trusted proxy concurrently, the network latency between the clients and the trusted proxy is assumed to be 0 ms. The trusted proxy and the storage server run on the same machine -it is

[9]We stress that our algorithm presentation above *does* lock the whole tree – this makes the proof slightly simpler, but the proof extends also to this higher level of granularity.

[10]Measured using iPerf tool[2].

the machine that is used as a trusted proxy in the initial set of experiments.

In both set of experiments, each bucket is configured to have four blocks of size 4 KB each. The default dataset sizes are 1 GB, i.e. 244,140 blocks and 13 GB, i.e. 3,173,828 blocks for real world and simulation based experiments, respectively. Additionally, the write-back threshold is set to $k = 40$ paths.

In our experiments, the clients issue concurrent read and write requests. Three parameters may affect the performance of the system: 1) the number of clients, 2) the scheduling of client requests, and 3) the network bandwidth. For 2), we consider an *adaptive scheduling of requests*, where each client sends the next request immediately after receiving the answer for the previous one. The requested blocks are selected from a uniformly distributed workload and each set of experiments uses the same workload[11].

The main metrics to evaluate the performance are *response time* and *throughput*. Response time spans the time period from initiating a client request until the time that this client receives a response. This metric shows how fast the system can handle client requests. Throughput is defined as the number of (concurrent) requests that the system answers per unit time. The goal is to achieve a low average response time while ensuring high throughput. To report reliable results, each set of experiments is run multiple times and the averages of the gathered results are presented with a 95% confidence interval. Some intervals are not clearly seen in Figure 8 due to their small sizes compared to the scale.

We also note that in order to calculate the experimental results in the steady state, the system is warmed up before taking any measurements. Warming up is achieved by the first 10% of the workload.

C. Experimental Results

1) Cloud-based TaoStore Evaluation: In this section, we vary different system parameters and study their effects on the performance of TaoStore by deploying it to a real world environment using AWS.

a) Effect of Concurrency: Figure 8(a) shows the effect of concurrency on TaoStore's average response time and throughput while varying the number of concurrent clients from 1 to 15. The left and right vertical axes represent throughput and response time, respectively.

With a single client, the response time is 55.68 ms, which leads to a throughput of 17.95 op/s. As the number of concurrent clients increases, the throughput also increases as long as the system can support more simultaneous operations. The system reaches its limit and stabilizes at a throughput of approximately 40 ops/s when the number of concurrent clients is 10. When the number of concurrent clients goes above 10, the clients generate more requests than the system can handle concurrently. In such a case, the clients experience increasingly worse performance in terms of response time although the performance of the system does not degrade in terms of throughput. Consider the case when the number of clients is 15. Although the system achieves approximately the same throughput at around 40 ops/s, the response time increases by 45% compared to the case with 10 concurrent clients. We observe that the network bandwidth is the main bottleneck in our experiments and it is the main reason for the observed behavior. Each path request results in transferring approximately 260-270 KBytes of data from the storage server to the proxy. Since the system handles 40 ops/s, the bandwidth utilization of the system is approximately 10.4-10.8 MBytes/s. Recall that the downstream network bandwidth is 11 MBytes/s, the system utilizes almost all the bandwidth and achieves its best throughput performance at around 40 ops/s.

To understand the system behavior with higher network bandwidth, we perform an additional set of experiments by running a proxy on another Amazon EC2 instance in the same datacenter where the storage server is located. The proxy runs on an m3.xlarge EC2 machine and we measure the bandwidth between the server and the proxy to be 125.25 MBytes/s. In this setting, the system achieves a throughput of 97.63 ops/s with an average response time of 102 ms when the number of clients is 10. The system performance increases dramatically with the increase in network resources, 149% increase in the throughput and 60% decrease in the response time.

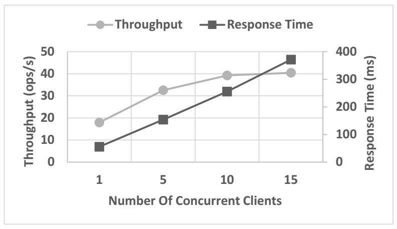
As a result of our experiments we observe that higher bandwidth can facilitate outstanding improvements in the system performance. Therefore, the bandwidth is one of the important issues for oblivious cloud storage systems in a realistic deployment setting as well as supporting concurrency and asynchronicity.

Please note that the default setting for the number of concurrent clients is 10 in the rest of our experiments unless otherwise stated.

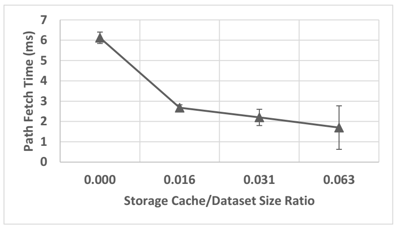
b) Caching at the Cloud Storage: Caching the top levels of the tree at the untrusted cloud storage eliminates a significant amount of the I/O overhead. Figure 8(b), 8(c) and 8(d) present the effects of applying caching in terms of response time, throughput, and path fetch time versus caching ratio. The caching ratio represents the amount of data cached in the cloud memory compared to complete dataset size. When there is no caching, the requested buckets in the path are fetched in 6.12 ms from the disk. When the caching is applied, the cached buckets are retrieved from the memory and the remaining buckets are fetched directly from the disk. Caching 1.6% of the dataset, approximately 16 MBytes, decreases path retrieval time from 6.12 ms to 2.68 ms. As the caching ratio increases, the time to fetch path decreases. When this ratio is 6.3%, the path is fetched in 1.7 ms. However, 3-4 ms performance improvement in data retrieval is not reflected in the overall system performance in terms of response time and throughput because of the network bandwidth limitations. As Figure 8(c) and 8(d) show, the system provides similar throughput and response time over varying caching ratios.

c) Impact of the Write-back Threshold: Recall that the write-back threshold k determines the number of paths that are

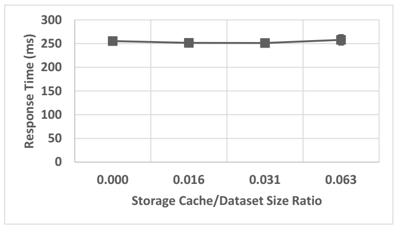
[11]Please note that the distribution of requested blocks does not affect the performance of TaoStore unlike ObliviStore.



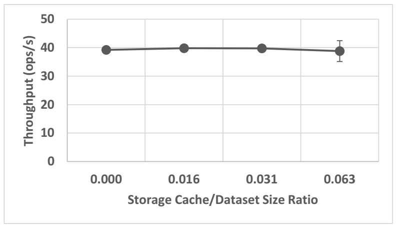
(a) Effect of Number of Concurrent Clients



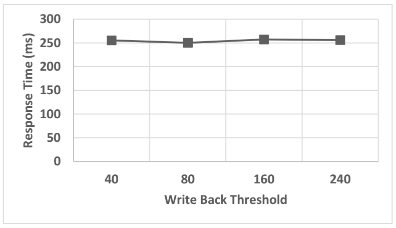
(b) Effect of caching at the untrusted server on average path fetch time from disk



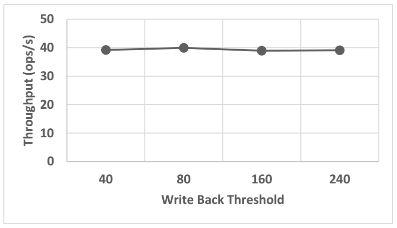
(c) Effect of caching at the untrusted server on response time



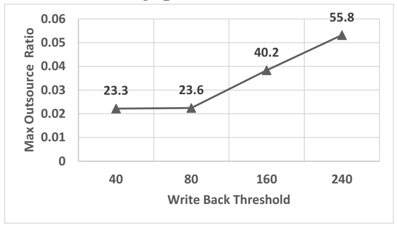
(d) Effect of caching at the untrusted server on throughput



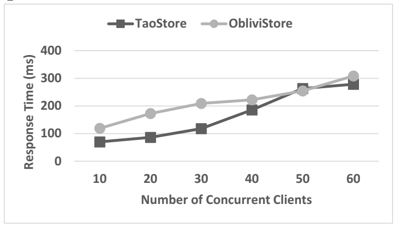
(e) Effect of write-back threshold on response time



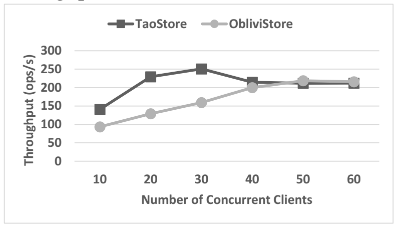
(f) Effect of write-back threshold on throughput



(g) Effect of write-back threshold on maximum outsource ratio (Data labels represent maximum utilized memory in MBytes)



(h) Effect of number of concurrent clients on response time



(i) Effect of number of concurrent clients on throughput

Fig. 8: TaoStore Performance Analysis

retrieved from the untrusted cloud storage before a write-back operation is initiated. A large k requires storing more data at the trusted proxy. However, this results in triggering less write-back operations and performing them in bigger batches. The effects of this parameter in terms of the average response time and throughput are demonstrated in Figure 8(e) and 8(f). As it can be seen in the results, there is no significant change in the performance with respect to k. This explicitly shows the design advantages of the non-blocking write-back mechanism, since the system performance is independent of the frequency of write-backs.

d) Memory and Bandwidth Overhead: TaoStore's memory overhead mostly depends on the write-back threshold k. In our experiments, we observe that the number of stored blocks in the stash usually does not exceed $2k$. When k equals 40, the stash usually does not contain more than 80 blocks, which requires approximately 320 KB in memory. Therefore, the stash memory overhead is a small constant, while the subtree uses more memory to store retrieved blocks from the untrusted storage. The overall memory usage for the trusted proxy is usually not more than 24 MB when $k = 40$ as shown in Figure 8(g), which has an approximate outsource ratio of 0.02. The *outsource ratio* is the ratio of maximum memory usage at the trusted proxy over dataset size. To answer one client query, the trusted proxy needs to fetch approximately 16 buckets, i.e., 256 KB. Increasing the flush trigger count results in using more memory at the trusted proxy; however, there is not much performance gain from increasing the write-back threshold. When $k = 240$, the trusted proxy uses a maximum of 55.8 MB memory, but achieves a throughput of 39.09 ops/s. The results show that TaoStore can deliver a good performance with a very low outsource ratio.

2) Comparison with Other Works: We now compare TaoStore with Path ORAM and ObliviStore to show how TaoStore can achieve high throughput and lower response times. The implementation of ObliviStore was provided by its authors[12] and we implemented our own version of Path ORAM. All experiments in this section are simulation based and have the same configuration.

Path ORAM provides relatively low response times of 63.63 ms with a corresponding throughput of 7.9 ops/s. Since Path ORAM does not support concurrency, it is not fair to compare

[12]We would like to thank the authors of ObliviStore for providing us the implementation graciously.

it directly with TaoStore. However, the results highlight the importance of providing concurrency for cloud storage systems (also highlighted in [4]).

Although ObliviStore is not secure over asynchronous networks and fails to provide complete access privacy when concurrent requests access the same item even over synchronous networks, the comparisons with ObliviStore aim to provide insights about TaoStore's performance while providing stronger security. Note that the simulation based experiments assume a 50 ms fixed round-trip network latency. Such an assumption prevents network bandwidth limitation issues. Once data is fetched from the disk drive, operations are executed in memory with delays on the order of 1 ms. The performance is affected mainly by the ORAM client side processing and data retrieval from the disk. Please note that since a uniformly distributed workload is used in the experiments, the probability for accessing the same ORAM blocks, which causes a slowdown for ObliviStore as highlighted in [4], is negligible.

Response times and throughput are compared for both systems in Figures 8(h) and 8(i), respectively. TaoStore and ObliviStore achieve their highest performances at 30 and 50 clients, respectively. When the number of clients is 30, TaoStore reaches a throughput of 250.79 ops/s with a response time of 117.91 ms. When the number of concurrent clients is 30, ObliviStore delivers a throughput of 159.35 ops/s with a response time of 209.07 ms. Hence, TaoStore achieves 57% high throughput with 44% lower response time. ObliviStore has performance issues against demanding applications due to its complex background shuffling and eviction operations (also pointed out in [4]). It deploys an internal scheduler to manage evictions and client requests but in contrast to TaoStore, the eviction process is not directly decoupled from the client request processing. The scheduler schedules a client request if the system has enough resources available. When the client request is scheduled, it acquires some amount of system resources and these resources are released once the eviction operations are completed. On the other hand, TaoStore can process client requests concurrently and asynchronously, and the write-back operations are decoupled from the client request processing. This allows TaoStore to continue processing client requests while one or more write-back operations are ongoing. With 30 concurrent clients, available resources are utilized aggressively to provide better performance in terms of throughput and response time. This explicitly demonstrates the design advantages of TaoStore compared to ObliviStore. If the number of concurrent clients goes above 30, Taostore's throughput shows a slight decline and the response time increases, due to the increased contention on processing units and I/O. TaoStore's performance plateaus after 40 clients with a throughput of 211-215 ops/s. ObliviStore's achieves its highest throughput of 218.56 ops/s with a response time of 254.45 ms at 50 clients.

In these experiments, a 13 GB dataset is used as in the experimental setup for ObliviStore [36]. In order to operate over a 13 GB dataset, TaoStore requires 15.9 GB physical disk storage in the untrusted cloud storage, while ObliviStore requires 42.9 GB. The difference in storage overhead is due to a significant number of extra dummy blocks ObliviStore requires [36], i.e., if a level in a partition is capable of storing up to x number of real blocks, the same level stores x or more dummy blocks. However, in tree ORAMs, dummy blocks are used to pad buckets if they contain a lower number of real blocks than their capacity. As also seen in the results, TaoStore is a lot less costly compared to ObliviStore in terms of required physical disk storage.

Our evaluations show that TaoStore handles flush and write-back operations better than ObliviStore, which leads to a high client request processing performance.

VI. Conclusion and Ongoing Work

TaoStore is a highly efficient and practical cloud data store, which secures data confidentiality and hides access patterns from adversaries. To the best of our knowledge, TaoStore is the first *tree-based* asynchronous oblivious cloud storage system. Additionally, we propose a new ORAM security model which considers completely asynchronous network communication and concurrent processing of requests. It is proven that TaoStore is secure and correct under this security model. Our experiments demonstrate the practicality and efficiency of TaoStore.

We are currently exploring extending TaoStore for fault-tolerance, since the system is vulnerable to multiple types of failures, including critically the failure of the proxy and the failure or inaccessibility of the untrusted public cloud. We are currently developing methods to use persistent local storage, i.e., disk, in the private cloud to overcome the failure of the proxy server. On the other hand, for the public cloud data, we are developing replication methods that span multiple clouds (possibility owned by different providers).

Acknowledgements

We would like to thank the anonymous reviewers for their constructive and helpful comments.

Cetin Sahin, Victor Zakhary, and Amr El Abbadi were partly funded by NSF grants CNS-1528178 and CCF-1442966. Huijia Lin was partially supported by NSF grants CNS-1528178 and CNS-1514526. Stefano Tessaro was partially supported by NSF grants CNS-1423566, CNS-1528178, and the Glen and Susanne Culler Chair.

References

[1] Amazon Web Services. https://aws.amazon.com/.

[2] iPerf - the TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/.

[3] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403, Miami Beach, Florida, Oct. 19–22, 1997. IEEE Computer Society Press.

[4] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15*, pages 837–849, Denver, CO, USA, Oct. 12–16, 2015. ACM Press.

[5] D. Boneh, D. Mazieres, and R. Popa. Remote oblivious storage: Making oblivious ram practical. MIT Tech-report: MIT-CSAIL-TR-2011-018, 2011.

[6] E. Boyle, K.-M. Chung, and R. Pass. Oblivious parallel RAM and applications. In E. Kushilevitz and T. Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 175–204, Tel Aviv, Israel, Jan. 10–13, 2016. Springer, Heidelberg, Germany.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15*, pages 668–679, Denver, CO, USA, Oct. 12–16, 2015. ACM Press.

[8] B. Chen, H. Lin, and S. Tessaro. Oblivious parallel RAM: Improved efficiency and generic constructions. In E. Kushilevitz and T. Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 205–234, Tel Aviv, Israel, Jan. 10–13, 2016. Springer, Heidelberg, Germany.

[9] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.

[10] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 62–81, Kaoshiung, Taiwan, R.O.C., Dec. 7–11, 2014. Springer, Heidelberg, Germany.

[11] K.-M. Chung and R. Pass. A simple oram. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/.

[12] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, Oct. 1971.

[13] J. Dautrich, E. Stefanov, and E. Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, San Diego, CA, Aug. 2014. USENIX Association.

[14] S. Devadas, M. Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In E. Kushilevitz and T. Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174, Tel Aviv, Israel, Jan. 10–13, 2016. Springer, Heidelberg, Germany.

[15] C. W. Fletcher, M. van Dijk, and S. Devadas. Towards an interpreter for efficient encrypted computation. In *Proceedings of the 2012 ACM Workshop on Cloud computing security, CCSW 2012, Raleigh, NC, USA, October 19, 2012.*, pages 83–94, 2012.

[16] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, pages 1–18, 2013.

[17] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[18] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[19] M. T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In M. Charika, editor, *21st SODA*, pages 1262–1277, Austin, Texas, USA, Jan. 17–19, 2010. ACM-SIAM.

[20] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP 2011, Part II*, volume 6756 of *LNCS*, pages 576–587, Zurich, Switzerland, July 4–8, 2011. Springer, Heidelberg, Germany.

[21] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 95–100, 2011.

[22] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious storage with low I/O overhead. *CoRR*, abs/1110.1851, 2011.

[23] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Y. Rabani, editor, *23rd SODA*, pages 157–167, Kyoto, Japan, Jan. 17–19, 2012. ACM-SIAM.

[24] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, San Diego, California, USA, Feb. 5–8, 2012. The Internet Society.

[25] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Y. Rabani, editor, *23rd SODA*, pages 143–156, Kyoto, Japan, Jan. 17–19, 2012. ACM-SIAM.

[26] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 199–213, San Jose, CA, 2013. USENIX.

[27] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. PHANTOM: practical oblivious computation in a secure processor. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13*, pages 311–324, Berlin, Germany, Nov. 4–8, 2013. ACM Press.

[28] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*, pages 341–358, San Jose, California, USA, May 17–21, 2015. IEEE Computer Society Press.

[29] T. Moataz, T. Mayberry, and E. Blass. Constant communication ORAM with small blocksize. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 862–873. ACM, 2015.

[30] N. I. of Standards and Technology. Advanced encryption standard (aes). Federal Information Processing Standards Publications - 197, November 2001.

[31] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303, El Paso, Texas, USA, May 4–6, 1997. ACM Press.

[32] B. Pinkas and T. Reinman. Oblivious RAM revisited. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 502–519, Santa Barbara, CA, USA, Aug. 15–19, 2010. Springer, Heidelberg, Germany.

[33] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, Washington, D.C., Aug. 2015. USENIX Association.

[34] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 571–582, 2013.

[35] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214, Seoul, South Korea, Dec. 4–8, 2011. Springer, Heidelberg, Germany.

[36] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267, Berkeley, California, USA, May 19–22, 2013. IEEE Computer Society Press.

[37] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *NDSS 2012*, San Diego, California, USA, Feb. 5–8, 2012. The Internet Society.

[38] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13*, pages 299–310, Berlin, Germany, Nov. 4–8, 2013. ACM Press.

[39] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *ESORICS 2006*, volume 4189 of *LNCS*, pages 49–64, Hamburg, Germany, Sept. 18–20, 2006. Springer, Heidelberg, Germany.

[40] X. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15*, pages 850–861, Denver, CO, USA, Oct. 12–16, 2015. ACM Press.

[41] P. Williams and R. Sion. Single round access privacy on outsourced storage. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 12*, pages 293–304, Raleigh, NC, USA, Oct. 16–18, 2012. ACM Press.

[42] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM CCS 08*, pages 139–148, Alexandria, Virginia, USA, Oct. 27–31, 2008. ACM Press.

[43] P. Williams, R. Sion, and M. Sotáková. Practical oblivious outsourced storage. *ACM Trans. Inf. Syst. Secur.*, 14(2):20, 2011.

[44] P. Williams, R. Sion, and A. Tomescu. PrivateFS: a parallel oblivious file system. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 12*, pages 977–988, Raleigh, NC, USA, Oct. 16–18, 2012. ACM Press.

The **storage server** SS is initialized with an array D of M items from T (which is kept as the state), exposes a *network* and an *adversarial* interface. It associates with every $\mathsf{bid} \in [M]$ a corresponding timestamp τ_{bid} – initially set to 0 – and operates as follows:

- At initialization, it outputs D at the adversarial interface.
- On input $\mathsf{op} = (\mathsf{bid}, u, \tau)$ at the *network interface*, the request is associated with a unique identifier id and $\mathsf{op} = \mathsf{op}_{\mathsf{id}}$ is added to the *input buffer*. The message $(\mathtt{input}, \mathsf{id}, \mathsf{bid}, u, \tau)$ is output at the adversarial interface.
- On input $(\mathtt{process}, \mathsf{id})$ at the adversarial interface, then $\mathsf{op}_{\mathsf{id}} = (\mathsf{bid}, u, \tau)$ is removed from the input buffer. We then set $v_{\mathsf{id}} = D[\mathsf{bid}]$ and if $u \neq \perp$, also sets $D[\mathsf{bid}] = u$ if $\tau_{\mathsf{bid}} < \tau$ (and update τ_{bid} to τ). The value v_{id} is added to the *output buffer* and returned at the adversarial interface.
- On input $(\mathtt{output}, \mathsf{id})$ at the adversarial interface, the value v_{id} is removed from the output buffer, and output at the network interface.

Fig. 9: The storage server functionality SS.

Appendix

A. *Security of Asynchronous ORAM Schemes*

This section develops a framework to analyze the security asynchronous ORAM schemes. We exercise this model to prove TaORAM secure.

Reactive systems. We consider a model of randomized interactive stateful reactive machines (sometimes simply called "algorithms"), which we only specify informally here, and which mimic the architecture running TaoStore. These machines have multiple interfaces, each with a given name.

The machines can activate at any time a *thread* by a certain input condition being met a certain interface (for example, a set of messages satisfying a certain condition have been input) and the corresponding messages are removed and input to the thread. During its execution, the thread can output messages at an interface, can set local variable and global variables (and can lock and unlock global variables), and can halt waiting for input messages to satisfy some condition to be re-started. Such threads can be run concurrently, and we do not make any assumptions about how thread executions are interleaved.

Such machines can then be combined with each other by connecting interfaces with the same name. (We can think of a combination of such machines as a network of machines, but also as a bigger machines.) Consistent with literature on cryptography and asynchronous systems, we do not assume a global clock: When a thread halts waiting for a message, it does not learn how long it has been waiting.

Asynchronous ORAM. An **asynchronous ORAM scheme** is a pair $\mathsf{ORAM} = (\mathsf{Encode}, \mathsf{OClient})$ consisting of the two following algorithms:

1) The **encoding algorithm** Encode on input a data set D (i.e., an array of N items from a set S), outputs a processed data set $\hat{D}$ and a secret key K. Here, $\hat{D}$ is an array of $M = M(N)$ elements from a set T.
2) The **ORAM client** $\mathsf{OClient}$ is initiated with the secret key K, as well as M and N. It maintain two interfaces: The **user interface** receives read/write requests (bid_i, u_i), where $\mathsf{bid}_i \in [N]$ is a logical address for the data set and $u_i \in S \cup \{\perp\}$ a data item. These requests are eventually answered by a value $v_i \in S$. The **network interface**, $\mathsf{OClient}$ issues server read/write requests of form $(\mathsf{bid}_j, u_j, \tau)$, where $\mathsf{bid}_j \in [M]$, $u_i \in T \cup \{\perp\}$, and $\tau \in \mathbf{N}$, and which are eventually answered with a value $v_i \in T$.

The (finite) sets S and T denote the data types of the items held by the ORAM data structure and the storage server, respectively. Formally, all algorithms take as input a security parameter λ in unary form, and the sets S and T may depend on this security parameter. We omit mentioning λ explicitly for ease of notation. We also stress that in contrast to our algorithm descriptions in the body of the paper, for notational compactness here we think of $\mathsf{OClient}$ as answering a single type of read-write operation – i.e., (bid, u) simply retrieves the value of block bid if $u = \perp$, and additionally overwrites it with u if $u \neq \perp$.

Our scheme TaORAM can naturally be expressed in this framework. Here, the set S would correspond to individual data items addressed by bid, whereas T would correspond to bit-strings representing encrypted blocks.

Adaptive security. Our security definition, which we refer to as *adaptive asynchronous obliviousness*, or aaob-security, is *indistinguishability* based. In contrast to existing security notions – which are typically non-adaptive – our definition allows for adaptive scheduling of operations and messages. In particular, we model the non-deterministic nature of scheduling messages in the communication between the server and the client by leaving the scheduling task to the adversary $\mathcal{A}$. To achieve this, the security game involves a *storage server* SS, which is initially given an array of M elements from some set T, and exposes a *network* interface and an *adversarial* interface. It operates as described in Figure 9. In particular, beyond its natural functionality at the network interface, the adversarial interface leaks the contents of read/write accesses and allows control of their scheduling.

For an asynchronous ORAM scheme $\mathsf{ORAM} = (\mathsf{Encode}, \mathsf{OClient})$ and an adversary $\mathcal{A}$, we define the experiment $\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$ as in Figure 10. We can then define the aaob-*advantage* of the adversary $\mathcal{A}$ against ORAM as

$$\mathsf{Adv}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A}) \Rightarrow \mathtt{true}\right] - 1 .$$

We stress that the adversary schedules concurrent operation pairs – previous operations do not need to have returned (and thus $\mathcal{A}$ has been notified) before other operations are scheduled by $\mathcal{A}$.

Experiment $\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$:

- Initially, a challenge bit $b \xleftarrow{\$} \{0,1\}$ is chosen uniformly at random.
- The adversary $\mathcal{A}$, given no input, outputs two data sets D_0, D_1, each with N items.
- Then, $(\widehat{D}, K) \leftarrow \mathsf{Encode}(D)$ is computed, and we give $\widehat{D}$ and K as initial inputs to the server SS and to the client OClient, respectively.
- After that, the adversary $\mathcal{A}$ communicates with the adversarial interface of SS. Also, the network interfaces of OClient and SS are connected with each other. Finally, at any point in time, $\mathcal{A}$ can output a pair of operations $(\mathsf{op}_{i,0}, \mathsf{op}_{i,1})$, and the operation $\mathsf{op}_{i,b}$ is forwarded to the user interface of OClient.
- When each operation terminates and a reply is given at OClient's user interface, the adversary $\mathcal{A}$ is going to be notified (however, it does *not* learn the result of the operation).[a]
- Finally, $\mathcal{A}$ outputs a guess b'. If $b = b'$, the experiment returns `true`, and `false` otherwise.

[a]Note that leaking which value is returned by the operation can lead to easy distinguishability.

Fig. 10: Experiment for aaob-security definition.

Definition 1 (ORAM Security)**.** *We say that am ORAM Protocol* $\mathsf{ORAM} = (\mathsf{Encode}, \mathsf{OClient})$ *is* aaob**-secure** *(or simply secure) if* $\mathsf{Adv}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$ *is negligible for every polynomial-time adversary* $\mathcal{A}$.

We note that aaob-security in particular implies[13] security according to Definition 1 in [36], which has adversaries issue a fixed sequence of operations with fixed timings.

B. Security of TaORAM

We now prove the following theorem, assuming that the underlying encryption scheme satisfies the traditional notion of (secret-key) IND-CPA security [18], [3].

Theorem 2 (Security)**.** *Assume that the underlying encryption scheme is IND-CPA secure, then TaORAM is secure.*

Proof (Sketch): The proof is more involved than for traditional, non-concurrent, ORAM schemes. We omit a complete formal proof for lack of space. However, we outline the main steps necessary for the formal argument to go through, which in particular explains the central role played by the sequencer.

Specifically, we note the following central properties of TaORAM:

[13]Formally speaking, their definition allows the choice of the scheduling of operations to be fixed according to some absolute clock. Following the cryptographic literature here we omit access to an absolute clock, and parties have only accesses to logical sequences of events. We note that [36] does not include a formal model.

- Every operation op to OClient results in the Processor immediately starting a thread retrieving the contents of exactly one fresh random tree-path $\mathsf{pid}_{\mathsf{op}}$ from the server. This is regardless of the type of operation issued, or whether fake.read is set or not. The adversary can then schedule OClient's requests as it wishes.
- The processor never replies to an operation *before* the whole contents of $\mathsf{pid}_{\mathsf{op}}$ have been received from the storage server, and never replies *after* the last path $\mathsf{pid}_{\mathsf{op}'}$ associated with an operation op′ preceding op in sequencer.queue is completely retrieved.
- The sequencer replies to an operation request op immediately after $\mathsf{pid}_{\mathsf{op}}$ and all paths $\mathsf{pid}_{\mathsf{op}'}$ associated with operations op′ preceding op in sequencer.queue have been completely retrieved.
- Write backs occur after a fixed number of paths have been retrieved, independently of the actual operations having been issued, and consists of fresh encryptions.

The above four items imply that the *communication* patterns are oblivious: The view of the adversary $\mathcal{A}$ in the experiment $\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$ does *not* depend on the actual choice of the challenge bit b, when the adversary cannot see the contents of the messages sent over the network. In particular, $\mathcal{A}$ can see explicitly the mapping between op and the path $\mathsf{pid}_{\mathsf{op}}$, and $\mathcal{A}$'s decision on when the contents of the path are given back to OClient completely determines the timings of the responses.

Given this, we note that the case $b = 0$ and $b = 1$ cannot be distinguished even given the contents of the messages and the storage server. To show this, the proof first replaces every encrypted block (either in a message or on the server) with a fresh encryption of a dummy block (e.g., the all-zero block). This does not affect the adversary's aaob advantage much by IND-CPA security of the underlying encryption scheme, and the fact that the adversary never sees the actual responses to its operations. Given now that the encrypted contents can be simulated and are independent of the actual operations issued, we can now apply the above argument showing that the actual access patterns are indistinguishable. ■

C. Histories, Linearizability, and Correctness

We note that security of an asynchronous ORAM scheme as defined above does not imply its correctness – one can just have the client do nothing (i.e., not sending any message to a server) and immediately reply requests with random contents, and have a secure scheme. For this reason, we handle correctness separately and show that our TaORAM satisfies very strong correctness guarantees, and in particular provides so-called *atomic* semantics of the underlying storage from a user-perspective. This means that every operation appears to have taken place atomically at some point between the request and the answer is provided. To formalize this notion, we follow the tradition of the literature on distributed systems and consistency semantics. We start with some definitions.

To reason about correctness, let us think of a variation of Experiment $\mathsf{Exp}^{\mathsf{aaob}}_{\mathsf{ORAM}}(\mathcal{A})$ defined above where the reply to each adversarial request is actually given back to the adversary, and moreover, we do not have a challenge bit any more. More formally, we define $\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{ORAM}}(\mathcal{A})$ as the following experiment, with no output:

Experiment $\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{ORAM}}(\mathcal{A})$:

- The adversary $\mathcal{A}$, given no input, outputs a data set D with N items.
- Then, $(\widehat{D}, K) \leftarrow \mathsf{Encode}(D)$ is computed, and we give $\widehat{D}$ and K as initial inputs to the server SS and to the client OClient, respectively.
- After that, the adversary $\mathcal{A}$ communicates with the adversarial interface of SS. Also, the network interfaces of OClient and SS are connected with each other. Finally, at any point in time, $\mathcal{A}$ can output an operation op_i, which is forwarded to the user interface of OClient.
- When each operation terminates and a reply is given at OClient's user interface, the adversary $\mathcal{A}$ is going to be notified and learns the outcome of the operation.

Recall that the client OClient processes requests of the form (bid_i, v_i), where v_i is either a data item (for an overwrite operation), or $v_i = \perp$ (for a read operation), and this operation is replied with a data item u_i. In an execution of the above experiment, we associate with every request a unique *operation identifier* $i \in \mathbf{N}$ in increasing order, with the goal of paring it with the corresponding reply.

A **history** Hist consists of the initial data set D, as well as a sequence of items of the form $\mathsf{req}_i = (\mathsf{bid}_i, v_i)$ and $\mathsf{rep}_i = u_i$, such that every occurrence of some item $\mathsf{rep}_i = u_i$ is preceded by a (unique) element $\mathsf{req}_i = (\mathsf{bid}_i, v_i)$ with the same identifier i. We say that a history is *partial* if there exists $\mathsf{req}_i = (\mathsf{bid}_i, v_i)$ without a corresponding $\mathsf{rep}_i = u_i$, and otherwise it is *complete*. An execution of $\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{ORAM}}(\mathcal{A})$ naturally generates a history at the user interface of OClient, where the sequence of requests and responses corresponds to the point in time in which they were given as an input to OClient by $\mathcal{A}$, and returned as an output to $\mathcal{A}$, respectively.

In a complete history Hist, we refer to the pair $(\mathsf{req}_i, \mathsf{rep}_i)$ as op_i (the i-th operation) and we say that op_i *precedes* op_j if and only if rep_i occurs before req_j. Also, we often write $\mathsf{op}_i = (\mathsf{bid}_i, u_i, v_i)$. We say that a complete history Hist is **linearizable** if there exists a total order $\leq_{\mathsf{lin}}$ over the operation identifiers such that: (1) If op_i precedes op_j, then $\mathsf{op}_i \leq_{\mathsf{lin}} \mathsf{op}_j$. (2) If $\mathsf{op}_i = (\mathsf{bid}_i, v_i, u_i)$, then either the largest $\mathsf{op}_j = (\mathsf{bid}_j, v_j, u_i)$ such that $\mathsf{op}_j \leq_{\mathsf{lin}} \mathsf{op}_i$ and $v_j \neq \perp$, if it exists, is such that $v_j = u_i$, or no such op_j exists and $D[\mathsf{bid}_i] = u_i$.

With the above definitions in place, we are ready to state the following definition.

Definition 2 (Correctness). *An asynchronous ORAM scheme* $\mathsf{ORAM} = (\mathsf{Encode}, \mathsf{OClient})$ *is* **correct**, *if for all adversaries $\mathcal{A}$ (even computationally unbounded ones) that deliver all messages, the history generated by* $\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{ORAM}}(\mathcal{A})$ *is complete and linearizable, except with negligible probability.*

D. Correctness Proof for TaORAM

We apply the above definition to TaORAM.

Theorem 3 (Correctness). *TaORAM is correct.*

Proof: For this analysis, we assume that memory never overflows, and thus the system will never crash or abort. (We discussed above that lack of memory overflows can be assumed without loss of generality.)

We show below that if $\mathcal{A}$ delivers all messages, then every history is complete at the end of the execution of $\mathsf{Exp}^{\mathsf{corr}}_{\mathsf{ORAM}}(\mathcal{A})$. The core of the proof is to show that the resulting complete history Hist is linearizable. This requires first defining the corresponding order $\leq_{\mathsf{lin}}$.

For every operation $\mathsf{op}_i = (\mathsf{bid}_i, v_i, u_i)$, there is a point in time t_i in which it takes effect in the global event sequence (we assume that every event is associated with a unique time). This is always within ANSWER-REQUEST in the execution of Item 3. In particular, an operation $\mathsf{op}_i = (\mathsf{bid}_i, v_i, u_i)$ takes effect when it is popped from the queue $\mathsf{request.map}[\mathsf{bid_i}]$. (Note that this may be within a thread running ANSWER-REQUEST for another operation op_j for which $\mathsf{bid}_j = \mathsf{bid}_i$.) We order two operations $\mathsf{op}_i = (\mathsf{bid}_i, v_i, u_i)$ and $\mathsf{op}_j = (\mathsf{bid}_j, v_j, u_j)$ so that $\mathsf{op}_i \leq_{\mathsf{lin}} \mathsf{op}_j$ if op_i takes effect before op_j. Clearly, if op_i precedes op_j, then $\mathsf{op}_i \leq_{\mathsf{lin}} \mathsf{op}_j$, since every operation takes effect between the request and the response.

During the execution of TaORAM, we can track the contents of the local storage, and we are going to prove the following invariant:

Invariant. At every point in time, there exists at most one value B_{bid} for the block bid in the local storage (sub-tree or stash). Moreover, this value is the latest value assigned to bid according to the "take-effect" order defined above (or the initial value, if no such value exists).

Note that before returning a value u for an operation on bid, we must have set the local value B_{bid} before returning $B_{\mathsf{bid}}[\mathsf{bid}]$, and thus the above implies that $\leq_{\mathsf{lin}}$ is a proper ordering to show that the history is linearizable.

To prove the invariant, we proceed by induction over steps that can modify the contents of the local storage. The invariant is true when the system has been initialized, and the client's local memory is empty. The following operations can modify the contents of the local storage (here, a pair $(\mathsf{bid}, B_{\mathsf{bid}})$ in the local storage simply denotes a pointer to block bid and the actual contents of the block).

1) A pair $(\mathsf{bid}, B_{\mathsf{bid}})$ is added to the local storage as part of some node w through processing of some path pid in Step 1 of ANSWER-REQUEST.
2) A pair $(\mathsf{bid}, B_{\mathsf{bid}})$ is deleted at Step 5 of WRITE-BACK because it is on a path pid written back to the server.
3) A pair $(\mathsf{bid}, B_{\mathsf{bid}})$ is moved to a new location (either in the tree or into the stash) when shuffling within FLUSH
4) A pair $(\mathsf{bid}, B_{\mathsf{bid}})$ is present in the local storage, and we assign B_{bid} to some new value v, in the third item of Step 3 of ANSWER-REQUEST.

Clearly, 3–4 do not violate the invariant. As for 2, if B_{bid} has been modified after it has been written to the server, then it will not be deleted due to the node timestamp being now higher than $v \cdot k$. If it is deleted, then no modification has occurred since the write-back has started, and thus the server holds the latest version.

The core of the proof is showing that 1 cannot violate the invariant, which we do next. In fact, we prove now that if at some time t^* the invariant has been true so far, and we now insert $(\mathsf{bid}, B_{\mathsf{bid}})$ as part of the contents of a node N, then this is the latest value of bid and no other value for bid appears in the local storage at this point in time t^*.

First off, if this is the initial value written by the Encode procedure into the server, and it gets written into node N, and $(\mathsf{bid}, B_{\mathsf{bid}})$ was never locally in node N, then the value of bid was never modified locally, because we need to retrieve it from the server at least once for the first change to take effect. Therefore, we can assume that $(\mathsf{bid}, B_{\mathsf{bid}})$ was already once earlier in the local storage at node N, either because it was written back from there (if this is not the initial value), or because we need to retrieve it at least once if this is the initial value and some modification has taken place. Now, consider the time $t \leq t^*$ at which $(\mathsf{bid}, B_{\mathsf{bid}})$ was in N for the last time. Note that if $t = t^*$, then the value would not be overwritten (as the node N is occupied in the local storage) and by the induction assumption this node holds the latest value. Therefore, assume that $t < t^*$, and we have two cases.

The first (and more difficult) case is that, at time t, $(\mathsf{bid}, B_{\mathsf{bid}})$ left N, and was possibly modified one or more times. In this case, we show that the local storage is *not* updated because the node N is already occupied with some pre-existing contents. The important observation here is that if there are one or more completed write-backs between t and t^*, the node N is never deleted after the write back completed. If it left N, then N was modified, and a write-back terminating after t would not delete N *unless* it just wrote this new contents of N back (or an even newer version). But this means that at that point we have already overwritten the contents of N *on the server* with something different than what received within pid (i.e., where in particular $(\mathsf{bid}, B_{\mathsf{bid}})$ would not be in N any more). Hence, the contents $(\mathsf{bid}, B_{\mathsf{bid}})$ of N received with pid must have been sent by the server before the new contents have been written (this is ensured by our server time stamping), and thus when this write-back completes, we have $\mathsf{pid} \in \mathsf{PathReqMultiSet}$, and hence N is left untouched and unmodified.

The second case is that, at time t, $(\mathsf{bid}, B_{\mathsf{bid}})$ was deleted after a successful write back completed. As this was the last time $(\mathsf{bid}, B_{\mathsf{bid}})$ ever appeared in N before t^* it cannot be that any operation to effect on bid between t and t^*, and thus the value re-covered with pid is the latest one.

We still need to show that every operation eventually terminates, and thus every history is eventually completed. We first show that the Processor Module replies to every request. Note that if all messages are delivered by $\mathcal{A}$, the wait instructions in READ-PATH always terminates, and the thread is waken up. Therefore, every retrieved path is eventually received by the client. Now there are two cases, for the thread executed for an operation accessing bid_i – either it results in a fake read or not, i.e., the flag fake.read returned by READ-PATH is either 1 or 0.

- *Case 1:* $\mathsf{fake.read} = 0$: Here, we know that the path P contains bid, and when executing ANSWER-REQUEST, either the entry in response.map for this operation has form $(\texttt{false}, x)$ for $x \neq \perp$, then the operation is answered right away in Step 2. Alternatively, if $x = \perp$, because the block is in the path P, this query must be replied later in Step 3.
- *Case 2:* $\mathsf{fake.read} = 1$. Then, this means that while executing READ-PATH in the main thread T, another thread T' has invoked READ-PATH for the same bid_i without returning $\mathsf{fake.read} = 1$, and thread T' has not yet gone through Step 3 in ANSWER-REQUEST. Now, there are two cases. Either T' will update the value for the current request in response.map in Step 3 of ANSWER-REQUEST before T goes though Step 2 in its own ANSWER-REQUEST, in which case T will return the value. Alternatively, if T goes through Step 2 first, the value will be output when T' goes through Step 3.

Finally note that the sequencer module may delay answering, but the above argument implies that the processor eventually answers all previous requests, and thus the sequencer will also eventually answer them all. ■