Joshua Gancher, Adam Groce, and Alex Ledger

Externally Verifiable Oblivious RAM

Abstract: We present the idea of *externally verifiable* oblivious RAM (ORAM). Our goal is to allow a client and server carrying out an ORAM protocol to have disputes adjudicated by a third party, allowing for the enforcement of penalties against an unreliable or malicious server. We give a security definition that guarantees protection not only against a malicious server but also against a client making false accusations. We then give modifications of the Path ORAM [15] and Ring ORAM [9] protocols that meet this security definition. These protocols both have the same asymptotic runtimes as the semi-honest original versions and require the external verifier to be involved only when the client or server deviates from the protocol. Finally, we implement externally verified ORAM, along with an automated cryptocurrency contract to use as the external verifier.

Keywords: Oblivious RAM, cryptocurrency

DOI 10.1515/popets-2017-0021
Received 2016-08-31; revised 2016-11-30; accepted 2016-12-01.

1 Introduction

Oblivious RAM (ORAM) protocols allow clients to store data on an untrusted server and to access it as if it was stored locally. The protocols guarantee that the server learns nothing about the client's data *or access pattern*, seeing only the amount of data and how frequently it is accessed. Originally proposed by Goldreich and Ostrovsky [5, 6, 8], in recent years a series of exciting results (e.g., [4, 9, 15, 16]) has drastically increased the efficiency with which ORAM can be achieved.

These results prove security in one of two models. The first is the *semi-honest* model, where the server is assumed to behave according to the specified protocol. Given this honest behavior, the security claim requires that no information about the client data or access pattern can be inferred by the server. The second model is that of *malicious* security, in which these guarantees must hold even if the server deviates from the prescribed protocol. Malicious security is certainly a stronger model, and it is probably more realistic in most settings. It has generally been achieved by first creating a semi-honest protocol and then adding verification procedures that will detect any deviation from the protocol that the server attempts to make.

Here we propose *externally verifiable* ORAM. In the standard proposed use of ORAM protocols, a client pays a server to provide storage in the cloud. If a malicious-secure ORAM protocol is used, the client gets complete assurance that their data and access patterns remain private. However, if server misbehavior is detected, the client has no redress other than to abort the protocol. Given that server misbehavior in this setting might often be aimed not at violating client privacy but instead at reducing costs by failing to store certain data (or by storing it unreliably, without necessary backups and safeguards), it may be very important that a client can show to a third party that the server has failed. Externally verifiable ORAM provides that capability.

Because one of the primary concerns is data loss, it is important to guarantee that the client can be compensated in the case of non-response from the server. Because it is impossible to prove non-response after the fact, the external verifier must participate in the protocol. Fortunately, we can minimize this burden — in our protocols, the verifier does *not* participate in honest executions. The verifier is only contacted when one party detects that the other has deviated from the protocol, at which point the protocol enters a second phase in which messages are routed through the verifier who is then able to confirm non-response.

We present three scenarios to illustrate how externally verifiable ORAM might be used.

Scenario 1 Client signs a contract with Server for remote storage at a given price. A penalty is specified, to be paid if Server becomes unable or unwilling to answer read/write queries correctly. If that does happen, Client goes to Judge and claims Server is not providing the specified data, at which point Server must respond with the requested data or is forced to pay the penalty.

This is the most straightforward use of our protocol. A standard government court probably does not

Joshua Gancher: Cornell University, work done partly while at Reed College, E-mail: jrg358@cornell.edu
Adam Groce: Reed College, E-mail: agroce@reed.edu
Alex Ledger: MIT Lincoln Laboratory, work done while at Reed College, E-mail: alex.ledger@ll.mit.edu

have the technical capability (or willingness) to serve as Judge in this scenario, since that requires real-time participation in the protocol. The contract would have to include an agreement to resolve disputes through arbitration and specify an arbitrator capable of this technical operation, but binding arbitration agreements are not unusual.

Scenario 2 Client purchases storage from Company, a large and trusted corporation. Company does not store Client's data directly, instead subcontracting with Server, who happens to have unused disk space at the moment. This allows administrators with spare capacity to put that capacity to work easily and reduces wasted resources. Client interacts directly with Server, but Company acts as a guarantor of Server's reliability, moderating disputes and penalizing Server if data is not reliably available to Client when requested.

Finding a way to use otherwise idle computational resources is potentially very useful, but it is hard to establish a functioning market in these services when the providers are small and have no reputation for reliability. Here an established company with a valuable reputation to protect serves as a guaranteed enforcer of reliability. This company can also serve to coordinate the many providers. If one administrator needs to reclaim their previously idle resources, the company can facilitate the transfer of the remote database to another server without disruption of the client's service.

Scenario 3 Client and Server reach an agreement as in Scenario 1. Instead of signing a legal contract, they create this "contract" as an automated entity in a cryptocurrency protocol like Ethereum that allows for automated execution of contracts. (See [17] for more information.) This contract serves the role of Judge above, automatically mediating disputes between Client and Server and penalizing Server where appropriate using currency held in escrow.

This is perhaps the most interesting scenario. It essentially circumvents the requirement of a third party. While technically many more than three parties are now involved (due to the nature of the cryptocurrency protocol), in practice this interaction can be carried out by any two parties on their own. The security guarantees of the cryptocurrency protocol ensure the community of cryptocurrency users will execute the automated contract as specified.

This community-as-verifier entity is potentially more trustworthy than any individual real world party could ever be. It is also easier to initialize than any real world contract, since no legal agreements are required. However, using an automated contract as the verifier is not without downsides. While a legal arbitrator has the authority to force payment of a penalty from the dishonest party's general property, a cryptocurrency contract can only force payment by holding currency in escrow. Also, as can be seen in our implementation, carrying out this computation is much slower and more expensive when done by an automated contract than when done by a single trusted party. Nevertheless, we think this idea is ideal for some situations and that these downsides can be reduced with future research.

1.1 Our contributions

Security definition In this paper we formalize *externally verifiable ORAM* (EVORAM), the functionality that is needed for a protocol to be used in the above scenarios. This is difficult because one of the server misbehaviors that we want to protect against is non-response. There is no way a client can prove to the verifier that they did not receive a response to a given message. To solve this problem, we divide our desired functionality into two phases. In Phase 1, the client interacts directly with the server, attempting to carry out a given operation. If the server fails to respond or responds incorrectly, the client can proceed to Phase 2, where communication is routed through the verifier, who can therefore confirm nonresponse. This means that full security is guaranteed, but the verifier does not participate during honest executions of the protocol.

Modified Path ORAM We then modify the Path ORAM protocol [15] to achieve this definition. Our modifications are reasonably straightforward, using standard techniques like Merkle trees, signatures, and counters to guarantee accuracy and freshness. We present this first as a simple example of what is needed to guarantee external verifiability. Even though the tools we use are standard, there are several subtle technical details that must be handled carefully. Our construction maintains the efficiency of Path ORAM.

Modified Ring ORAM We then present an EVORAM modification of Ring ORAM. While Path ORAM requires $O(\log n)$ communication per access (where n is the database size), Ring ORAM requires only $O(1)$ communication per access in *online* communication, with other communication deferred to flexibly-scheduled offline operations, and we want to maintain this efficiency in the externally verifiable construction. Ring ORAM

also uses a more complex underlying data structure. These two concerns collectively mean that while we can use (mostly) the same basic tools, the construction is significantly more complicated. Here too we maintain the asymptotic running time of the original protocols. (The one exception is that we require more communication in our version of Ring ORAM *only* when a deviation from the protocol occurs and the verifier needs to be involved. Since in practice resort to the verifier is primarily a deterrent, rather than part of normal operation, we think this downside is minimal.)

Malicious-secure Ring ORAM As a stepping stone to our EVORAM construction, we present what we believe is the first modification of Ring ORAM that provably achieves malicious security with no efficiency loss.

Stronger security guarantees We show that our constructions have useful security properties beyond what is required of the EVORAM definition. In particular, Scenario 3 requires that security be maintained with a semi-honest verifier, and this is the case in our protocols. Similarly, we show that if the verifier is malicious and collaborating with the server, the EVORAM security guarantee is lost, but the original ORAM guarantee is still maintained, meaning that trust in the verifier can be limited. This is most important in a setting like Scenario 2.

Implementations Finally, we provide two implementations of our modified Path ORAM protocol. The first is a standard implementation for three parties. It achieves EVORAM with only 1% to 3% increased bandwidth overhead compared to Path ORAM. Our second implementation uses an Ethereum contract as the verifier, allowing two parties to sign and execute an enforced contract for storage without an external arbitrator. In this implementation regular accesses are just as fast as in the previous implementation, but the verification that occurs when one party misbehaves is significantly slower and has a small monetary cost associated with it. All implementations are released under open-source licenses.

We believe that the additional functionality in our definition, as well as the demonstrated practicality of achieving it, go a long way towards making ORAM protocols useful in a variety of real-world situations.

1.2 Outline of the Paper

Section 2 discusses prior work on ORAM protocols and notions analogous to externally verifiable security in other contexts. Section 3 presents our definition of externally verifiable ORAM and discusses some of the subtleties involved in choosing the definition. Sections 4 and 5 are on Path and Ring ORAM, respectively. These sections begin with a brief overview of the original semi-honest protocol, explain how that protocol can be made secure in the malicious setting, then finally show how to modify the protocol to achieve our externally verifiable functionality. Section 6 describes our implementation of externally verifiable Path ORAM. We discuss future directions for this work in Section 7.

2 Background

Oblivious RAM seeks to allow a client to outsource storage to an untrusted server. This can trivially be done by encrypting the data, but that requires that the client download the entire database for decryption every time they wish to access the data. The database can instead be divided into blocks, with each block encrypted separately under the client's key. While this greatly increases efficiency, since only a single block need be downloaded for each access, the server now knows which block is requested/changed with each client interaction, and that access pattern can disclose private information, either about what computation the client is performing or about the contents of the data itself. ORAM protocols ensure complete privacy for the client by hiding both the contents of the data *and* the access pattern.

Goldreich and Ostrovsky [5, 6, 8] first proposed ORAM and gave a protocol that achieved security. However, their protocol resulted in a factor of $O(\sqrt{n})$ increase in the required bandwidth compared to unsecure access, where n is the database size. A series of works improved on this construction. A major milestone was the proposal by Shi *et al.* to structure the ORAM storage as a binary tree [12]. That basic innovation inspired a string of ever-more-efficient protocols in recent years. (For example, see [4, 9, 15, 16].)

Of particular interest to this work are Path ORAM [15] and Ring ORAM [9]. Path ORAM was an early and simple example of the binary tree framework in use, and it decreased the required overhead to $O(\log n)$. Ring ORAM improved on the efficiency of Path ORAM, allowing for $O(1)$ blowup in *online* bandwidth. (Total

bandwidth required is still $O(\log n)$, but most communication happens in the background between accesses.) Other works have increased efficiency further, but these two protocols are the ones we adapt to create externally verifiable ORAM protocols. We leave creating externally verifiable versions of more recent protocols to later work.

The idea of externally verifiable ORAM is new in this paper. Previous work has introduced the idea of *verifiable oblivious storage* [1]. (Here "oblivious storage" is used to mean an ORAM protocol where the server can perform computation, rather than simply storing data. We do not make that distinction in this work and refer to such protocols as ORAM protocols.) The verifiability in question though refers only to the ability of the client to verify the server's honesty. It would not, for example, allow for the enforcement of a contract promising storage, since a malicious client could falsely claim server misbehavior, and a verifier would be unable to adjudicate the claim.

Externally verifiable security definitions do exist in other areas of research. For example, Stadler introduced publicly verifiable secret sharing, where parties outside of the secret sharing protocol can confirm that it has been carried out correctly [14]. There is also a line of work on *optimistically fair exchange* (e.g., [2] and [3]), which adds the same sort of external verifiability to protocols for exchanging digital goods. We suspect that the advent of smart contracts could be applicable to some of these works. Care must be taken, however, since the third parties considered in optimistic fair exchange are in general allowed to maintain secret state.

In the setting of outsourced storage, Shah, Swaminathan, and Baker [11] also give a similar functionality. They show a way that outsourced storage can be "audited," allowing an external arbitrator to confirm whether or not the server has lost some of the client's data. The motivation here is identical to ours. However, this was not built on top of ORAM's privacy protection — while data privacy was ensured against the auditor, the server itself had full access to the data and access pattern.

3 Externally Verifiable ORAM

We now present our definition of externally verifiable ORAM. Our goal here is to allow the external verifier $\mathcal{V}$ to arbitrate disputes, verifying that the server $\mathcal{S}$ is or is not properly carrying out the operations requested by the client $\mathcal{C}$. We want to protect against a malicious server who is attempting to misbehave, either by altering data or simply by hiding the fact that some data has been lost. We also want to protect against a malicious client who attempts to falsely claim server misbehavior. We assume the verifier is trusted. (We present the definition with a fully honest verifier. In reality we could weaken the definition to allow for a semi-honest verifier, and we discuss this alteration later.) We are at the moment concerned with security in the standard model; proving universal composability for our scheme is left to further work.

Because we are worried about a server that simply loses data and becomes unable to answer queries, we must allow a server to be punished for simple lack of response to client requests. However, in a two-party protocol the client can never prove after the fact that the server stopped responding.[1] One obvious solution is to route communication through the verifier, who can confirm that the client's request was sent and that the server did not respond. This, however, requires the verifier to participate in every access. Instead, we divide our protocol into two phases. In the first, the client sends requests directly to the server. During honest interaction, this first phase is all that occurs, and read/write operations are performed as expected. However, if the client detects unexpected behavior (including nonresponse) from the server, they can continue to Phase 2 of the protocol, which asks the verifier to mediate the operation. During this phase, the verifier can detect cheating from either party.

We formalize this desired behavior through the functionality in Figure 1. During the first phase of the ideal functionality, $\mathcal{C}$ submits a database operation and $\mathcal{C}$ and $\mathcal{S}$ submit $vrfy_{\mathcal{C}}$ and $vrfy_{\mathcal{S}}$ respectively. These are booleans specifying whether to involve the verifier in the operation's execution. Sending no for these values corresponds to honest behavior during Phase 1 of a protocol execution, and under honest behavior the access terminates successfully at that point. Either player has the power to deviate in Phase 1, which then forces the protocol to Phase 2, where they can again either behave honestly or deviate (represented by the $fail_{\mathcal{C}}$ and $fail_{\mathcal{S}}$

1 To see this formally, consider a client $\mathcal{C}$ who has transcript t of a successful protocol execution, and client $\mathcal{C}'$ who has transcript t' of a protocol execution that was identical up to a point where the server stopped responding. Because t' is computable given t, any "proof" of server misbehavior that $\mathcal{C}'$ could present to $\mathcal{V}$ (which must be a function of just t') can also be forged by $\mathcal{C}$ and is therefore unconvincing.

values in the functionality). If they deviate, the access ends unsuccessfully, and the cheating party is known by the verifier $\mathcal{V}$. If both parties behave honestly at this phase, the verifier learns only that an access has occurred.

The details of this functionality have been chosen very carefully, and we note that some seemingly equivalent choices actually introduce problems when one attempts to create secure protocols. For example, we considered a definition where the client had the option to use or not use the verifier on each access, essentially splitting Phase 1 and Phase 2 into two separate access calls, allowing the client to repeat a call with the verifier included any time the initial access failed. However, such a definition causes problems. We find that in order to construct secure protocols, one must structure the definition so that an honest client *always* must proceed to Phase 2 when Phase 1 fails. Failure to do so opens the client up to privacy-violating attacks.

We then require, through a standard simulation-based definition, that an externally verifiable ORAM protocol provide this functionality. We limit ourselves, though, to security against malicious clients or servers, and we require security for a party to hold only when it behaves honestly. In the ideal functionality, honest behavior by $\mathcal{C}$ means sending $vrfy_{\mathcal{C}} = fail_{\mathcal{C}} = \mathsf{no}$ (and similarly for honest $\mathcal{S}$).

Ideal World In this world a client $\mathcal{C}$ and server $\mathcal{S}$ (and a verifier $\mathcal{V}$, always assumed to be honest) interact with the functionality $\mathcal{F}$ and with an environment $\mathcal{Z}$. $\mathcal{V}$, being honest, always forwards any output received from $\mathcal{F}$ to $\mathcal{Z}$ unchanged. At the end of all poly(λ) accesses (where λ is the security parameter), $\mathcal{Z}$ outputs a bit. Define the random variable $\textsc{Ideal}_{\mathcal{C},\mathcal{S},\mathcal{Z}}(\lambda)$ to be this bit.

Real World In this world $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{V}$ communicate directly with each other and with the environment $\mathcal{Z}$. $\mathcal{V}$ always behaves as specified by the protocol and forwards all output to $\mathcal{Z}$. At the end of all accesses, $\mathcal{Z}$ outputs a bit. Define the random variable $\textsc{Real}_{\mathcal{C},\mathcal{S},\mathcal{Z}}(\lambda)$ to be this bit.

Definition 1 Fix a protocol Π. We say that the honest client $\hat{\mathcal{C}}$ is the one that carries out exactly the operations that $\mathcal{Z}$ requests and reports its output to $\mathcal{Z}$. In the real world $\hat{\mathcal{C}}$ always runs the protocol as specified. In the ideal world $\hat{\mathcal{C}}$ always sends $vrfy_{\mathcal{C}} = fail_{\mathcal{C}} = \mathsf{no}$. In both worlds, $\hat{\mathcal{C}}$ forwards all output (and nothing else) to $\mathcal{Z}$. Similarly, the honest server $\hat{\mathcal{S}}$ carries out the protocol honestly in the real world, always sends $vrfy_{\mathcal{S}} = fail_{\mathcal{S}} = \mathsf{no}$ in the ideal world, and reports its output to $\mathcal{Z}$.

We say an *externally verifiable ORAM scheme* is secure against a malicious server if for all probabilistic polynomial-time real world servers $\mathcal{S}$, there exists an ideal world simulator $\mathsf{Sim}_{\mathcal{S}}$, such that for all non-uniform, polynomial-time environments $\mathcal{Z}$, there exists a negligible function *negl* such that

$$|\Pr[\textsc{Real}_{\hat{\mathcal{C}},\mathcal{S},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\hat{\mathcal{C}},\mathsf{Sim}_{\mathcal{S}},\mathcal{Z}}(\lambda) = 1]| \leq negl(\lambda).$$

Similarly, we say it is secure against a malicious client if for all probabilistic polynomial-time real world clients $\mathcal{C}$, there exists an ideal world simulator $\mathsf{Sim}_{\mathcal{C}}$, such that for all non-uniform, polynomial-time environments $\mathcal{Z}$, there exists a function *negl* such that

$$|\Pr[\textsc{Real}_{\mathcal{C},\hat{\mathcal{S}},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathsf{Sim}_{\mathcal{C}},\hat{\mathcal{S}},\mathcal{Z}}(\lambda) = 1]| \leq negl(\lambda).$$

We say the scheme is *secure* if it is secure against both a malicious client and a malicious server. ♢

4 Path ORAM

In Section 4.1, we give an overview of semi-honest Path ORAM. Section 4.2 presents the modification for malicious security, while Section 4.3 gives our construction for externally verifiable security. Appendix A presents a proof for the security of our construction.

4.1 Semi-honest Path ORAM

Semi-honest Path ORAM is a simple ORAM protocol in which data on the server is stored in a binary tree, called the *ORAM tree* [15]. Each node of this tree is called a *bucket*, which is a collection of Z blocks. The client must also hold a small number of blocks locally in the *stash*.

An invariant must hold such that each block is mapped to a uniformly random leaf node of the ORAM tree, and blocks not in the stash must reside on the path in the ORAM tree corresponding to their leaf node. This mapping of blocks to leaf nodes is held in a structure called the *position map*. The *path associated to a block* B refers to the unique path on the ORAM tree which starts at the root node and ends at the leaf node associated to B, according to the position map. The position

Functionality $\mathcal{F}$

Setup:
Upon receiving the database D from the client, notify $\mathcal{S}$ that the setup operation has occurred and the size of the database $|D|$. If $\mathcal{S}$ returns ok, send ok to $\mathcal{C}$. If $\mathcal{S}$ returns abort, send abort to $\mathcal{C}$. The below steps are only valid if $\mathcal{S}$ sends ok.

Upon receiving an access $(op, ind, data, vrfy_{\mathcal{C}})$ **from** $\mathcal{C}$**:**
Notify $\mathcal{S}$ that an access is occurring.

Upon receiving $vrfy_{\mathcal{S}}$ **from** $\mathcal{S}$**:**
If $vrfy_{\mathcal{C}} = vrfy_{\mathcal{S}} =$ no, then process the requested operation $(op, ind, data)$ on behalf of $\mathcal{C}$, and send $\mathcal{C}$ the requested data. At this point, the access is completed.
Otherwise, send vrfy to $\mathcal{C}$ and $\mathcal{S}$.

Upon receiving $fail_{\mathcal{C}}$ **from** $\mathcal{C}$**:**
If $fail_{\mathcal{C}} =$ yes, send $\perp$ to $\mathcal{C}$, and send $\text{cheat}_{\mathcal{C}}$ to $\mathcal{V}$. At this point, the access is completed.

Upon receiving $fail_{\mathcal{S}}$ **from** $\mathcal{S}$**:**
If $fail_{\mathcal{S}} =$ yes, then send $\perp$ to $\mathcal{C}$, and send $\text{cheat}_{\mathcal{S}}$ to $\mathcal{V}$. At this point, the access is completed.
Otherwise, process the requested operation $(op, ind, data)$ on behalf of $\mathcal{C}$, and send $\mathcal{C}$ the requested data. Send success to $\mathcal{V}$. At this point, the access is completed.

Fig. 1. The externally verifiable ORAM functionality $\mathcal{F}$. vrfy, ok, and other similarly-formatted terms represent arbitrary agreed-upon constants meant to convey particular messages.

map can either be held in full by the client, or can be stored recursively in another Path ORAM instance.

When a block is read from the server, the client requests all blocks along the path associated to the desired block. Then, the requested block is remapped to another uniformly random leaf, and the entire path is rewritten back to the server from the stash, subject to the invariant.

Since to read or write a block in Path ORAM requires communicating all blocks along a path, Path ORAM has $O(\log n)$ bandwidth blowup, where n is the database size.

4.2 Malicious-Secure Path ORAM

Stefanov *et al.* extend semi-honest Path ORAM to provide integrity for every access when interacting with an untrusted server [15]. They achieve integrity by extending the role of the ORAM tree to also function as a Merkle tree, using a collision resistant hash function H. Each bucket of the ORAM tree also stores

$$H(b_1||\ldots||b_Z||h_1||h_2),$$

where $b_1, \ldots, b_Z$ are the blocks stored in the bucket, and h_1 and h_2 are the hashes of the left and right child. If the node is a leaf node, $h_1 = h_2 = 0$. We call this an *augmented* Merkle tree, to indicate that each node of the tree holds data, and not only the leaf nodes.

With this construction, the client only stores the hash held at the root of the Merkle tree. When reading a path from the server, the server also sends the appropriate hashes so that the client can recompute the root of the Merkle tree using their downloaded data. We call this collection of hashes a *Merkle proof*. For Path ORAM, a Merkle proof for a path consists of the hashes of sibling buckets along that path. We use ReconstructRoot(P, MP) to denote the root recomputed using the data P along a path and the corresponding Merkle proof, MP.

If the root does not recompute to the correct value, the client can conclude the integrity of the data has been violated. When rewriting the path, the client may use the same Merkle proof to recompute the new Merkle tree root.

This addition to Path ORAM is intuitively secure with a malicious server, and is outlined in Figure 2. If the position map is being stored recursively in another Path ORAM instance, the above process would be carried out for every level in the recursion. Ren *et al.* show that it suffices to only carry out the above process on the top level of recursion which holds data, and carry out a simpler authentication scheme on the lower levels which hold the position map [10]. We do not consider this extension here, but it is likely to be compatible with our externally verifiable version of Path ORAM.

Malicious Secure Path ORAM requires an additional $O(\log n)$ hashes to be communicated for each read due to the Merkle proof. Hence, adding integrity to Path ORAM does not affect the asymptotic bandwidth blowup compared to unauthenticated Path ORAM.

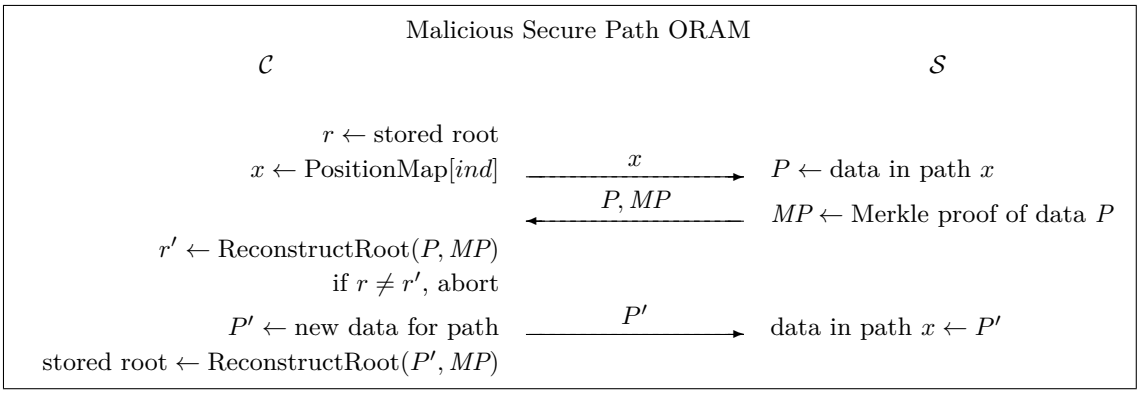

Fig. 2. An honest execution of Malicious Secure Path ORAM.

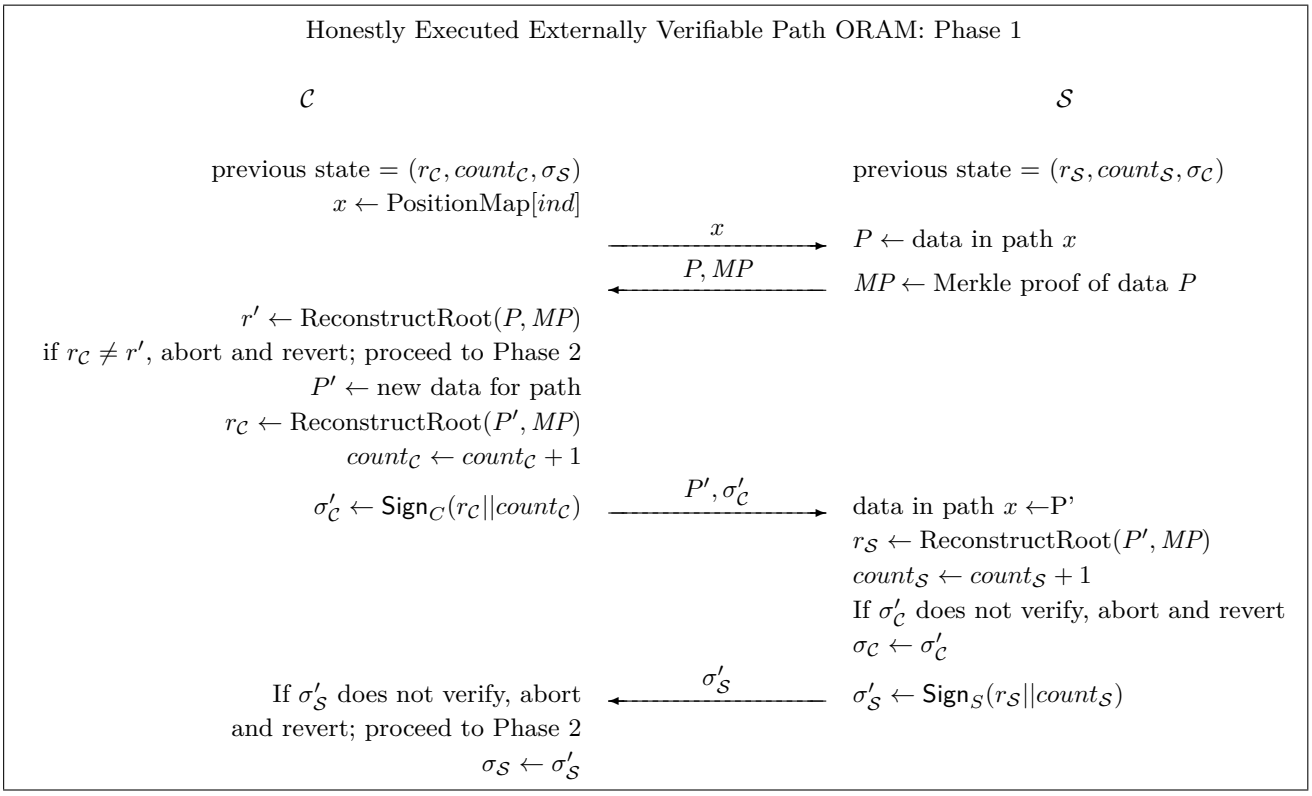

Fig. 3. An honest execution of Phase 1 of Externally Verifiable Path ORAM. The tuple $(r_\mathcal{C}, count_\mathcal{C}, \sigma_\mathcal{S})$ is acquired by the client from a previous access. Likewise, the server already holds $(r_\mathcal{S}, count_\mathcal{S}, \sigma_\mathcal{C})$. If either side chooses to abort, they "revert" their state, setting all values back to their state at the start of the execution before proceeding to Phase 2.

4.3 Externally Verifiable Path ORAM

We adapt this malicious-secure variant of Path ORAM into a protocol that is secure in the externally verifiable setting by using two standard tools. We require first that a counter be maintained that increments after each access to guarantee freshness. During setup, the client sets the counter to zero and in addition to sending the initial database also sends a signature on the counter and the root of the Merkle tree. The sever then responds with its own signature of the same values. After each access the counter is incremented and new signatures are exchanged, signifying agreement on the state of the database.

An access of externally verifiable Path ORAM consists of two phases. In Phase 1, the client attempts to interface directly with the server, as in Figure 3. If the client aborts during this operation, the client enters Phase 2. (If the server aborts during Phase 1, the client also aborts.)

In Phase 2, the client sends a request to the verifier to oversee the access, using the same access tuple $(op, ind, data)$ as in Phase 1. Phase 2 is similar to Phase 1, but each message is sent through the verifier; this is detailed in Figure 4. At the end of Phase 2, the verifier will output either $\mathsf{cheat}_{\mathcal{C}}$, $\mathsf{cheat}_{\mathcal{S}}$, or $\mathsf{success}$.

In this specification (and all others in this paper), all steps must happen in order. The verifier ignores any unexpected messages, and assumes any messages that are misformatted or not received in some specified amount of time are incorrect and indicate cheating by the sender. An output of $\mathsf{cheat}_{\mathcal{S}}$ favors the client (i.e., the server was detected to be cheating), while an output of $\mathsf{cheat}_{\mathcal{C}}$ favors the server. Once the verifier reaches an output command, no further commands for that access may occur; the verifier reverts to its initial state.

We assume that if the verifier outputs $\mathsf{cheat}_{\mathcal{S}}$, the client is notified and aborts the protocol. Similarly, if the verifier outputs $\mathsf{cheat}_{\mathcal{C}}$, the server aborts the protocol. Thus, we do not analyze a verified ORAM access that continues if the verifier outputs either of these messages.

Phase 2 We now formally define the client and server sides of Phase 2 of the protocol. This is initiated by a request from the client to the verifier comes in the form of $(root_{\mathcal{C}}, count_{\mathcal{C}}, \sigma_{\mathcal{S}})$ where $root_{\mathcal{C}}$ is the root hash of the Merkle tree, $count_{\mathcal{C}}$ is the state counter, and $\sigma_{\mathcal{S}}$ is the digital signature of the root and counter signed by the server. In particular, $\sigma_{\mathcal{S}}$ should equal $\mathsf{Sign}_{\mathcal{S}}(root_{\mathcal{C}}||count_{\mathcal{C}})$. If the signature in this initial message is valid, the verifier will forward $count_{\mathcal{C}}$ to the server.

The server responds analogously with $(root_{\mathcal{S}}, count_{\mathcal{S}}, \sigma_{\mathcal{C}})$. If $count_{\mathcal{C}}$ is one below the server's current count, the server should roll the database back to its state before the previous access, decrementing $count_{\mathcal{S}}$ to match $count_{\mathcal{C}}$, before sending this message. If $count_{\mathcal{C}}$ is more than one step older than $count_{\mathcal{S}}$, the server should not decrement — sending a (properly signed) counter that is more than one step ahead of the client's counter is proof of client misbehavior.

Assuming the server sends back a matching counter value, the protocol now proceeds identically to Phase 1. The only difference is that the client and server send messages to the verifier, which then forwards them to the other party (after the correctness checks listed in Figure 4).

The ability of the server to roll back the database is perhaps the most subtle part of this protocol. Because the server receives the new $\sigma_{\mathcal{C}}$ value before the client receives the new $\sigma_{\mathcal{S}}$, the server could fail to send the final message in Phase 1, and then have a signature on a one-higher count value during Phase 2, meaning that the verifier must not see the client as cheating if its claimed count value is one behind that of the server. However, this opens a possible attack — the client could run Phase 2 with a counter outdated by one increment, to which the server could not respond properly. To prevent this attack, the server must be able to roll back the database by one step. Since the previous state of the database differs from the current state in only one path, the required extra storage is $O(\log n)$, trivial compared to the overall size of the database.

Externally verifiable Path ORAM adds the exchange of a constant amount of hashes to every ORAM access (compared to the malicious-secure variant), so the cost is minimal.

See Appendix A for the proof that this protocol is secure.

4.4 Additional security properties

Two of the three scenarios we discussed at the beginning of this paper require slightly stronger security guarantees than the standard definition we have given. These modifications are straightforward, and we discuss each below. While we present these discussions here, they apply not only to the Path ORAM modification discussed

Verifier $\mathcal{V}$

Upon receiving $(root_{\mathcal{C}}, count_{\mathcal{C}}, \sigma_{\mathcal{S}})$ **from client:**
If $\sigma_{\mathcal{S}}$ does not verify correctly, output $\mathsf{cheat}_{\mathcal{C}}$. Otherwise, send $(\mathsf{verify\ request}, count_{\mathcal{C}})$ to server.
Upon receiving $(root_{\mathcal{S}}, count_{\mathcal{S}}, \sigma_{\mathcal{C}})$ **from server:**
If $\sigma_{\mathcal{C}}$ does not verify correctly, output $\mathsf{cheat}_{\mathcal{S}}$. If $count_{\mathcal{S}} \geq count_{\mathcal{C}} + 2$, output $\mathsf{cheat}_{\mathcal{C}}$. Otherwise if $count_{\mathcal{C}} \neq count_{\mathcal{S}}$, output $\mathsf{cheat}_{\mathcal{S}}$.
Upon receiving x **from client:**
Forward x to server.
Upon receiving (Path P**, Merkle proof** MP**) from server:**
Use data P with MP to reconstruct root r^*. If $r^* \neq root_{\mathcal{C}}$, output $\mathsf{cheat}_{\mathcal{S}}$. Verifier forwards (P, MP) to client.
Upon receiving (Path P'**,** $(root'_{\mathcal{C}}, count'_{\mathcal{C}}, \sigma'_{\mathcal{C}})$**) from client:**
Use data P' with MP to reconstruct new root r'.
If $\sigma'_{\mathcal{C}}$ does not verify correctly, output $\mathsf{cheat}_{\mathcal{C}}$. If $count'_{\mathcal{C}} \neq count_{\mathcal{C}} + 1$, output $\mathsf{cheat}_{\mathcal{C}}$. If $r' \neq root'_{\mathcal{C}}$, output $\mathsf{cheat}_{\mathcal{C}}$.
Forward P' and $(root'_{\mathcal{C}}, count'_{\mathcal{C}}, \sigma'_{\mathcal{C}})$ to server.
Upon receiving $(root'_{\mathcal{S}}, count'_{\mathcal{S}}, \sigma'_{\mathcal{S}})$ **from server:**
If $\sigma'_{\mathcal{S}}$ does not verify correctly, output $\mathsf{cheat}_{\mathcal{S}}$. If $count'_{\mathcal{S}} \neq count'_{\mathcal{C}}$, output $\mathsf{cheat}_{\mathcal{S}}$. If $r' \neq root'_{\mathcal{S}}$, output $\mathsf{cheat}_{\mathcal{S}}$.
Forward $(root'_{\mathcal{S}}, count'_{\mathcal{S}}, \sigma'_{\mathcal{S}})$ to client. Output $\mathsf{success}$.

Fig. 4. The externally verifiable Path ORAM verifier $\mathcal{V}$.

above, but also to the Ring ORAM modification that follows.

A semi-honest verifier In Scenario 3, where the verifier is implemented as an automated cryptocurrency contract, the verifier is guaranteed to behavior correctly but the information it sees is visible publicly and can be analyzed by those with malicious intent. That is, the verifier is semi-honest. Our protocols remain secure when the verifier is only semi-honest, assuming we alter the security definition slightly. Specifically, the verifier in our protocol sees the client and server *count* variables, which let the verifier learn how many accesses have occurred since the last verifier involvement. This is information that is not visible in the ideal world. The ideal functionality must be modified so that the verifier sees the count of the operation whenever the verifier is used. Then our protocols are secure with a semi-honest verifier (even if the verifier is collaborating with a malicious client or server). This reduction in security is quite minor, since ORAM protocols in general already leak the number of accesses. Security against a semi-honest verifier with collaborating malicious client or server is exactly what is needed to allow the verifier to be replaced by an automated cryptocurrency contract.

A malicious verifier In Scenario 2, Client relies on well-known Company to enforce a contract that guarantees storage. Company can outsource this storage to Server. This allows administrators of small systems with spare resources (but without the reputation to sell services directly) to act as Server and put those resources to use. However, Client might not want to fully trust Company, even if they are indeed more reliable than Server.

Fortunately, the trust needed in Company is minimal. One need not rely on Company for privacy protection, only for ensuring reliability. In particular, even if Company and Server are *both* malicious and are cooperating, Client still has the normal protection enjoyed under the standard ORAM definition, meaning that data and access pattern privacy are maintained. Only the additional utility of our external verifiability property is lost. This can easily be seen simply by noting that everything the server and verifier see in our protocols is either seen also in standard Path/Ring ORAM or is a computable function of those values.

5 Ring ORAM

We now present a construction for externally verifiable Ring ORAM. Section 5.1 gives an overview of Ring ORAM. Section 5.2 presents our construction for malicious-secure Ring ORAM, which utilizes authenticated encryption. Section 5.3 extends this construction to the externally verifiable setting, and Section D presents a proof of security.

5.1 Semi-honest Ring ORAM

Ring ORAM is an improvement of Path ORAM where instead of reading all blocks along a path every access, a single block per bucket is read, along with a path of

encrypted metadata [9]. In addition to each bucket storing Z blocks, each bucket also stores S dummy blocks. Whole buckets (meaning Z blocks out of the bucket) are still read and written in Ring ORAM, but much less frequently than Path ORAM.

Ring ORAM preserves the same invariant as Path ORAM: each block is mapped to a leaf node, and each block must be stored in a bucket along the path from the corresponding leaf node to the root of the ORAM tree.

When the client reads a block, they first complete a metadata scan over the path in question. They then decrypt the metadata and infer a set of offsets for each block along the path, where an offset i specifies that the ith block of the bucket is requested. This operation is called ReadPath. All of the blocks requested along the path, except the actual block that the client wants, are dummy blocks. The client then decrypts all blocks downloaded and only keeps the real block. Dummy blocks are each used only once; if a bucket along a path has been involved in more than S ReadPath operations, that bucket is now "used up". The number of ReadPath operations a bucket has been involved in is that bucket's *count*. At the end of ReadPath, all buckets with a count at least S are read to the client's stash, and rewritten from the stash. This rewriting operation is called an EarlyReshuffle.

Each bucket contains metadata that specifies where the Z real blocks are (along with other small pieces of metadata). The block size in Ring ORAM is set so that each block is larger than the collection of metadata read during ReadPath. Because of this, Ring ORAM is not performant for small block sizes.

Periodically, an *eviction* happens. In an eviction, a path is (deterministically) selected, and all buckets along that path are read and rewritten. This operation is called EvictPath. The operation to read an individual bucket to the client's stash is called ReadBucket, and the operation to write to an individual bucket is called WriteBucket.

Write operations are, in a sense, postponed. When a client wishes to write to a block, they instead store the new data in their local stash (and access the corresponding path, as if a read was occurring). Unless an EarlyReshuffle occurs, this data will not be written until an EvictPath operation allows the relevant block to be updated. (EvictPath is set to happen often enough that the size of the client's stash is bounded.)

Ring ORAM with XOR The ReadPath operation may be further optimized by having the server XOR all of the requested blocks together, and send the XORed value to the client. The client can reconstruct the dummy blocks, and hence recover the single non-dummy block from the XORed value. This construction gives a way to read a block securely where the only online communication is a single block. Thus, the online communication of Ring ORAM with XOR is a constant multiple of the block size. (Overall communication is still $O(\log n)$, but most can be performed offline.) Using the XOR optimization requires the client to store randomness associated with each dummy block, but this can, like other information, be stored recursively in another ORAM instance.

5.2 Malicious-Secure Ring ORAM

As a step on the way to externally verifiable ORAM, we contribute what we believe to be the first malicious-secure variant of Ring ORAM that does not reduce the asymptotic efficiency of the protocol.

In order to provide correctness, authenticity, and freshness, we use an authenticated encryption scheme AE. Each block stored on the server (both real and dummy) is stored as $\mathsf{AE.Enc}(c_i||p_i||b_i)$, where b_i is the block data, c_i is a *freshness counter* incremented each time b_i is written, and p_i is a *position index*, an encoding of the position in the ORAM tree of the block (i.e., $B||O$, where B is the bucket's unique identifier and O is the offset in the bucket).

The freshness counters for all blocks are stored on the client side in a data structure we call the FreshnessMap, which maps a block to a counter. When the client reads a block that decrypts successfully, the block is verified for correctness by checking that the read c_i agrees with the counter stored in the PositionMap, and that the read p_i corresponds to the correct position in the ORAM tree. The client is guaranteed authenticity by the block decrypting successfully.

Metadata for Ring ORAM also needs to be authenticated. For metadata, an augmented Merkle tree may be used in the same way as malicious-secure Path ORAM. We call this the *metadata tree*, or MT. Each internal node of MT is equal to $H(M_i||h_\ell||h_r)$, where M_i is the encrypted metadata in the ith bucket, and h_ℓ and h_r are the left and right children hashes of MT, respectively. The leaf nodes of MT are equal to $H(M_i)$. The client stores the current root r_M of the metadata tree.

Whenever the client begins a Ring ORAM operation (ReadPath, EarlyReshuffle, or EvictPath), the client

first requests from the server a path P_M of metadata. The server sends P_M along with a Merkle proof MP_M consisting of all sibling hashes for P_M. The client then locally reconstructs the metadata root r' using P_M and MP_M; if $r' \neq r_M$, the client aborts.

Each operation proceeds as follows:

1. The client requests a path P_M from MT, according to the ORAM access desired. In return, the server sends P_M along with a Merkle proof MP_M. Using the client's previously stored root r_M of MT, if ReconstructRoot$(P_M, MP_M) \neq r_M$, the client aborts.
2. The client runs the corresponding Ring ORAM operation. Each time block b_i is written to the server, c_i is incremented by one. If at any point a block contains an incorrect c_i or p_i, or decryption fails, the client aborts.
3. At the end of the access, the client sends an updated path of metadata P'_M to the server according to the metadata changed in the previous step; the client in turn updates its stored root r_M.

The above described scheme is malicious-secure. Additionally, this construction is compatible with Ring ORAM with XOR: the client can store locally all information needed to compute an authenticated encryption of a dummy block, so that they can recover a real block encrypted with authentication from an XORed path. We give a proof of malicious security in Appendix B.

As a speedup, the above scheme can be modified so that metadata is also verified using authenticated encryption in a manner similar to the ORAM data. (We use the Merkle tree in this construction in order to lead into externally verifiable Ring ORAM.)

Efficiency Overall, the additions above do not increase the space or communication complexity compared to semi-honest Ring ORAM.

Server storage increases only by a small constant amount (c_i, p_i, and the constant overhead for authenticated encryption) for each block.[2] Client storage is expanded to hold the freshness counters for each block. This additional per-block data can be stored recursively on the server similar to the Position Map, resulting in a constant blowup in client storage [15]. If the client uses the XOR technique, then they also need to store the randomness associated with each dummy block. This data may also be stored recursively.

Communication is increased by a Merkle proof MP_M for metadata communicated with each Ring ORAM operation. The size of this Merkle proof is dominated by the size of the path of metadata P_M required in the semi-honest Ring ORAM protocol.

5.3 Externally Verifiable Ring ORAM

We now describe how to make Ring ORAM secure in the externally verifiable setting. We note first that simply using the same techniques used for Path ORAM would result in $O(\log n)$ online bandwidth, removing the advantage that Ring ORAM (with XOR) offers in the first place. (This is because reading a block would require receiving a Merkle proof of that block's authenticity.) We instead use two separate types of overlapping authentication. The first is a set of Merkle trees: we use an augmented Merkle tree MT for the metadata tree, a (standard) Merkle tree BT for each bucket, and an augmented Merkle tree OT for the overall ORAM tree, where the "data" in each node is the root of the corresponding bucket tree. Combined with these Merkle trees, we also store in each block a counter and position index as we did in the malicious-secure variant. Each block is encrypted using authenticated encryption, as above. The data structures used for verifiable Ring ORAM are outlined in Figure 5.

The Merkle tree authentication and the authentication from authenticated encryption are used at different times. Metadata is always verified using the MT Merkle tree. When ReadPath is run, the resulting block is verified by successful decryption. For EarlyReshuffle and EvictPath operations, the OT Merkle tree (and the bucket trees) are used to verify data. The Merkle trees are updated whenever the data they authenticate is changed. Because successful decryption cannot be confirmed by the verifier, Phase 2 must revert to using the Merkle tree authentication process. This means that Phase 2 is a $O(\log n)$-communication operation. This is unfortunate, but given that the existence of Phase 2 is really just a deterrent — honest parties would never conduct this operation (and even malicious parties have no incentive to force Phase 2) — we believe this is a minor concern.[3]

2 Technically, c_i grows with $O(\log Q)$, where Q is an upper bound on the number of rewrites on any given block. With a practical block size (say, a few kilobytes), the space required for each c_i will be dominated by the size of each block.

3 One might be tempted to use signatures from both client and server on each block to avoid the $O(\log n)$ efficiency for Phase 2,

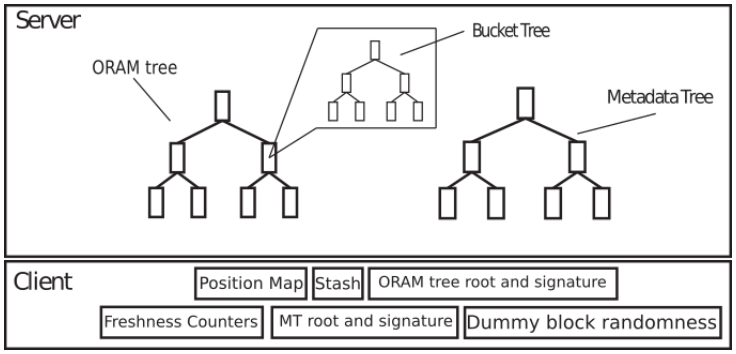

Fig. 5. The relevant data structures for externally verifiable Ring ORAM. All client-side storage except for the signed Merkle tree roots may be stored recursively.

We now outline the construction of externally verifiable Ring ORAM in more detail. Below we present high-level descriptions of exactly what changes need to be made to the semi-honest Ring ORAM operations. Full step-by-step descriptions of an honest execution of each operation can be found in in Appendix C.

Phase 1 Modifications here use authenticated encryption to authenticate blocks during ReadPath and Merkle trees for all other authentication. Counters and signatures for current values of the roots of OT and MT must be maintained. Specific details are stated below. We assume without explicit statement that any time one party receives a signature, block to decrypt, or Merkle proof, it is verified and if verification fails the receiving party aborts.

1. At the beginning of each operation, the client requests a metadata path P_M and receives both P_M and a Merkle proof MP_M of its correctness. At the end of the operation, the client sends the new path data to update MT as well as a signed copy of the new root, along with an (incremented) counter. The server responds with a signature of its own on the root and counter. (Note that this happens in all operations, meaning that the counters for MT and OT will not in general be equal.)
2. In the ReadPath operation, the client verifies the received block by checking successful decryption. (If not using the XOR variant, this is done for all blocks along the path.)
3. In EvictPath and EarlyReshuffle operations, the client receives data from particular blocks and a Merkle proof that those blocks are correct. (This proof includes a Merkle proof that a given bucket tree root is correct, followed by a Merkle proof that that root is correct based on the shared root of OT.) The client then writes new information to these blocks, recomputes the root of OT, increments its counter, and sends a signature of both to the server, who responds with a signature of their own.

Phase 2 This phase always consists of an EvictPath operation, moderated by the verifier. EarlyReshuffle and ReadPath can be conducted as part of EvictPath — they simply leave most data unchanged. The verifier independently confirms all signatures and outputs cheating messages if any do not verify correctly. More specifically:

1. Client submits signed roots and counters for OT and MT. Server responds with matching roots and counters. As with Path ORAM, the verifier accepts if the roots match, and the server is expected to be able to roll back OT, MT or both by one counter value. If the roots don't match, the verifier outputs a cheat message as in the Path ORAM variant.
2. A path P_O of data and path P_M of metadata is forwarded from the server to the client through the verifier, along with Merkle proofs MP_O and MP_M for these data.
3. Client uses P_O and P_M to call EvictPath as in semi-honest Ring ORAM, in order to obtain rewritten paths P'_O and P'_M. Client uses the above Merkle proofs to compute new roots r'_O and r'_M for the two Merkle trees.
4. Client sends new data P'_O and P'_M to verifier, along with client's signature of the new Merkle tree roots.

but this does not work. In particular, during Phase 2 the client must be sent the entire path anyway, in order to hide what block is requested. The XOR technique cannot be used because the verifier does not have the randomness needed to reconstruct the dummy blocks (and could not verify its accuracy if the client shared it).

Verifier $\mathcal{V}$

Upon receiving $(root_{T,C}, count_{T,C}, \sigma_{T,S})$ **for** $T \in \{O, M\}$ **from client:**
If $\sigma_{T,S}$ does not verify correctly using the corresponding root and counter for either $T \in O, M$, output $\mathsf{cheat}_{\mathcal{C}}$. Otherwise, send $(\mathsf{verify\ request}, count_{\mathcal{C}})$ to server.

Upon receiving $(root_{T,S}, count_{T,S}, \sigma_{T,C})$ **for** $T \in \{O, M\}$ **from server:**
For $T \in \{O, M\}$: If $\sigma_{T,C}$ does not verify correctly using the corresponding root and counter, output $\mathsf{cheat}_{\mathcal{S}}$. If $count_{T,S} \geq count_{T,C} + 2$, output $\mathsf{cheat}_{\mathcal{C}}$. Otherwise if $count_{T,C} \neq count_{T,S}$, output $\mathsf{cheat}_{\mathcal{S}}$.

Upon receiving x **from client:**
Forward x to server.

Upon receiving (Path P_T, **Merkle proof** MP_T**) for** $T \in \{O, M\}$ **from server:**
For $T \in \{O, M\}$: Use data P_T with MP_T to reconstruct root r^*_T. If $r^* \neq root_{T,C}$, output $\mathsf{cheat}_{\mathcal{S}}$.
If the above succeeds for $T \in \{O, M\}$, verifier forwards (P_M, MP_M, P_O, MP_O) to client.

Upon receiving (Path P'_T**,** $(root'_{T,C}, count'_{T,C}, \sigma'_{T,C})$**) for** $T \in \{O, M\}$ **from client:**
For $T \in \{O, M\}$: Use data P'_T with MP_T to reconstruct new root r'_T. If $\sigma'_{T,C}$ does not verify correctly, output $\mathsf{cheat}_{\mathcal{C}}$. If $count'_{T,C} \neq count_{T,C} + 1$, output $\mathsf{cheat}_{\mathcal{C}}$. If $r'_T \neq root'_{T,C}$, output $\mathsf{cheat}_{\mathcal{C}}$.
If the above succeeds for $T \in \{O, M\}$, forward P'_T and $(root'_{T,C}, count'_{T,C}, \sigma'_{T,C})$ for $T \in \{O, M\}$ to server.

Upon receiving $(root'_{T,S}, count'_{T,S}, \sigma'_{T,S})$ **for** $T \in \{O, M\}$ **from server:**
For $T \in \{O, M\}$: If $\sigma'_{T,S}$ does not verify correctly, output $\mathsf{cheat}_{\mathcal{S}}$. If $count'_{T,S} \neq count'_{T,C}$, output $\mathsf{cheat}_{\mathcal{S}}$. If $r'_T \neq root'_{T,S}$, output $\mathsf{cheat}_{\mathcal{S}}$.
If the above succeeds for $T \in \{O, M\}$, forward $(root'_{T,S}, count'_{T,S}, \sigma'_{T,S})$ for $T \in \{O, M\}$ to client.
Output $\mathsf{success}$.

Fig. 6. The externally verifiable Ring ORAM verifier $\mathcal{V}$. *root*, *count*, and σ variables refer to the roots of the Merkle trees, the corresponding counters, and the signatures thereof. Each variable has two subscripts, the first (O or M) indicating which tree the variable corresponds to, and the second ($\mathcal{C}$ or $\mathcal{S}$) denoting the party that computed the value.

5. Server responds with server's signatures of new Merkle tree roots.

The corresponding verifier can be seen in Figure 6.

Analysis Our additions to malicious Ring ORAM to make it externally verifiable do not cause any asymptotic overhead during honest execution. A Merkle proof for values in MT is only sent when a path from MT is being read as well, so this is a constant-factor increase. Similarly, Merkle proofs for values in OT are only sent when a path from OT is being sent. Counters, signatures, and the blowup from authenticated encryption are constant-size additional values. Phase 2 is the only potentially longer operation, causing $O(\log n)$ communication. This would only be done given malicious behavior (or hardware failure).

6 Implementation

We implemented the externally verifiable Path ORAM protocol of Section 4.3 in two ways. First, we implemented the client, server, and verifier protocols in C as would be used in Scenarios 1 or 2 described in our introduction. Second, as described in Scenario 3, we implemented an autonomous Ethereum contract that could act as the verifier, allowing two parties to interact with enforced penalties without the need for a third involved party acting as verifier. These implementations are both available online under open-source licenses.[4] We discuss the results of each implementation below.

Standard Verifier Our first implementation, using a standard real-world party as the verifier, requires roughly the same bandwidth as standard malicious-secure Path ORAM during regular interactions. The only additional bandwidth needed is for our protocol's added signatures (and a couple control bytes specifying that this is a Phase 1 interaction). This additional bandwidth is small and constant. We ran tests on databases of various sizes and find an overhead increase of exactly 259B per database access, regardless of database size. In our limited experiments on small databases, this ranged from 5.6% overhead (on a 262KB database) to 3.1% overhead (on a 134MB database). Extrapolation to a 1TB database would give an overhead of under 2%. We expect careful optimization could reduce this overhead somewhat.

A verified (Phase 2) interaction in this implementation requires exactly twice as much bandwidth (assuming all connections have equal latency and bandwidth).

4 https://github.com/gancherj/evoram

The same data is transmitted — the only change is that it is now transmitted from sender to verifier to receiver rather than directly from sender to receiver. This allows for punishment of very minimal gaps in service. The verifier, for example, can demand near-immediate responses from the server and penalize delays on a sliding scale. A server that experiences some small downtime or traffic beyond capacity can be penalized to a small extent,[5] while a server that loses data entirely (i.e., doesn't respond to a request correctly even after a long time) can face a very harsh penalty.

Ethereum Verifier We also provide a second implementation that uses an automated cryptocurrency contract to replace the third party verifier. We do this using Ethereum [17]. We stress that Ethereum is a cutting-edge cryptocurrency that has shown automated contracts to be realistic, but also that it is still in its early phases and is under active development. We view our results here as a demonstration of what is possible, and we expect that our precise measurements will be quickly out of date as the underlying cryptocurrency technology improves, whether that is through the improvement of Ethereum or through the introduction of other cryptocurrencies.

The implementation of our verifier protocol as an Ethereum contract was straightforward. The Ethereum project includes a language, Solidity, that is similar to ordinary scripting languages. All blockchain-specific implementation details, such as how the contract interacts with the Ethereum protocol, are abstracted away from the programmer; the structure of the smart contract closely mirrors the abstract verifier in Figure 4. The corresponding client and server implementations in C could also be written by a programmer lacking detailed knowledge of blockchain technology or Ethereum. The data being passed to the contract had to be converted into a standard format expected by Ethereum, but the actual "sending" of the data to the contract was handled using well-developed, open-source libraries. The contract measures time by checking the current progress of the blockchain, which proceeds at a stable pace. The contract cannot continue running autonomously; it only responds to messages. Therefore the client must ping the contract when an unacceptable amount of time has passed without a server response, at which point the contract checks the time and then penalizes the server (and vice versa for penalizing the client).

The implementation was not without challenges. Ethereum is still in its earliest stages and has a small user base. This meant that running times on public blockchains were highly variable, as sometimes a message was not processed on the first block step after it was submitted. Because of this, we ran our tests on a local simulation. (We also ran several experiments on the public testnet and mainnet, and found consistent results.) Using this method allows us to measure gas usage realistically, but does not allow us to measure timing data. The time complexity of each verified access is dominated by the number of rounds and intrinsic properties of the blockchain instance (such as average block time, and speed of the underlying gossip protocol). Given current blockchain parameters in Ethereum, this would be on the order of a small number of minutes, fast enough to enforce a contract that penalizes the server for losing data, but not fast enough to penalize the server for temporary failures of service quality.

The unverified (Phase 1) regular accesses do not depend on the verifier, and so are unchanged, remaining as efficient as under the first implementation. Bandwidth required for a verified (Phase 2) access is also largely unchanged relative to the first implementation.

We focused on measuring the cost of using the contract in a Phase 2 disputed access. Here one verified access costs the equivalent of roughly $0.33 on a 10TB database.[6] Again, this seems entirely reasonable for enforcing penalties for data loss, which we expect would be much greater than $0.33, but too expensive to enforce small micropayments as penalties for faulty service. The dependence on database size is $O(\log n)$, since gas cost is proportional to the length of the path being sent. See Table 1 for complete results.

We also fix database size and measure the dependency of cost on block size. (This is important because larger blocks are needed to enable recursive ORAM storage.) For block size b, we would expect $O(b \log(n/b))$ cost, and this is what we find. A 100MB database with 1KB block size gives a verification cost of $0.38. See Table 2 for full results.

Of course, these measurements are of completed verified accesses where client and server both honestly execute the protocol. In the event of nonresponse, the speci-

5 The only practical floor on how small a disruption can be punished is the reliability of the network. Sometimes network traffic is dropped and resent or otherwise delayed, and the time the verifier waits before declaring non-response must be sufficient to avoid blaming these delays on the server.

6 The contract could be devised to charge this cost to the server, the client, or any combination of the two.

DB size	Height	Total Gas	US Dollar equivalent
10MB	15	1077799	$0.18
1GB	22	1325838	$0.23
100GB	29	1632924	$0.29
10TB	35	1864503	$0.33

Table 1. Costs of external verification (Phase 2) using an Ethereum contract with varying database size. Block size is 96B (encrypted), with 5 blocks per bucket. Cost is the average over two runs. The US Dollar equivalent is relative to exchange rates as of November 28, 2016 ($8.74 per unit of ether).

Height	Encrypted Block Size	Total Gas	US Dollar equivalent
19	96	969879	$0.17
17	288	1221259	$0.21
15	1056	2145429	$0.38

Table 2. Costs of external verification (Phase 2) using an Ethereum contract with varying block size. Database size is 100MB (unencrypted), with 5 blocks per bucket. All costs are the average of two runs. The US Dollar equivalent is relative to exchange rates as of November 28, 2016 ($8.74 per unit of ether).

fied time must pass before penalizing one party. A failed verified access might also be much cheaper, since less of the contract's code is being executed.

Ethereum enforces a *gas limit* on the total amount of computation that can be done in a single block. Our verification costs are below this limit even for extremely large databases, but if many transactions of this complexity were being run simultaneously, collisions in the same block would cause degraded performance. However, in a scenario where Ethereum is used that widely its protocol would trigger an increase in the gas limit. This is a general issue with the scalability of blockchains; future iterations of Ethereum and similar platforms will undoubtedly work towards a more scalable blockchain.

Furthermore, certain new security concerns are raised in the setting of smart contracts. For example, nothing prevents a malicious client from carrying out a denial-of-service attack against the server, and using this as a proof of the server's non-response. Indeed, any escrow held by the contract is simultaneously a bug bounty for the contract itself; c.f. the infamous DAO hack [13]. One must also carefully assign responsibility for the cost of verified accesses. For example, if the server is responsible for the entire cost the client can force all transactions to Phase 2, imposing high costs on the server. (Whether this is in the client's interest depends on the setting.) While crucial, we consider these kinds of attacks as a separate issue from protocol design.

Whether a cryptocurrency contract will become fast and cheap enough to penalize intermittent server down time remains, we believe, an open question. Future externally verifiable ORAM protocols might be more carefully optimized for use with a cryptocurrency contract, but we expect the main avenue for improvement to be the underlying cryptocurrency technology. Order-of-magnitude improvements to the time and cost required to execute contracts are entirely possible.

7 Conclusion and Future Work

We have proposed what we believe is a useful definition, strengthening the guarantees of ORAM protocols in a way that allows for use in some practical situations that might have otherwise proven challenging. We then gave protocol constructions and implementations that show this definition can be achieved in reasonably efficient ways. However, much more remains to be done. Below, we outline several of the directions we feel are most interesting.

More efficient protocols We show the feasibility of externally verifiable ORAM by finding verifiable versions of the existing Path and Ring protocols. However, these are no longer the most efficient protocols known. We would love to see protocols that matched the efficiency of more recent standard ORAM constructions (e.g., [4, 16]).

Cryptocurrency improvements As mentioned above, we expect the state of the art in cryptocurrencies to change in the coming future. As a result, an implementation of our verifiable ORAM protocols over the next iteration of smart contract technology is likely to drastically improve the usability and cost of our system, as well as the time it takes to perform a verified access.

Automated verifiers Finally, we expect that the advent of autonomous third parties trusted for correctness (i.e., smart contracts) is likely to have interesting applications in other areas of security and privacy. In particular, we believe their use here could possibly be adapted to replace verifiers in optimistic fair exchange protocols and other related work. This can largely be enabled by using zero knowledge proofs of knowledge in order to facilitate manipulation of private data (see, for example, Hawk [7]).

References

[1] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography–PKC*, pages 131–148. Springer, 2014.

[2] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 7–17. ACM, 1997.

[3] N. Asokan, V. Shoup, and M. Waidner. *Optimistic fair exchange of digital signatures*, pages 591–606. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[4] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *IACR Theory of Cryptography Conference–TCC*, pages 145–174. Springer, 2016.

[5] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Symposium on Theory of Computing–STOC*, pages 182–194. ACM, 1987.

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

[7] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *University of Maryland and Cornell University*, 2015.

[8] R. Ostrovsky. Efficient computation on oblivious RAMs. In *Symposium on Theory of Computing–STOC*, pages 514–523. ACM, 1990.

[9] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: practical improvements to oblivious RAM. In *USENIX Security Symposium*, pages 415–430, 2015.

[10] L. Ren, C. W. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-RAM. In *High Performance Extreme Computing Conference–HPEC*, pages 1–6. IEEE, 2013.

[11] M. A. Shah, R. Swaminathan, and M. Baker. Privacy-preserving audit and extraction of digital contents. Technical report, HP Lab No. HPL-2008-32, 25 April, 2008.

[12] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Advances in Cryptology–ASIACRYPT*, pages 197–214. Springer, 2011.

[13] E. G. Sirer. Thoughts on the dao hack. http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/, 2016.

[14] M. Stadler. Publicly verifiable secret sharing. In *Advances in Cryptology—EUROCRYPT*, pages 190–199. Springer, 1996.

[15] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Computer & Communications Security–CCS*, pages 299–310. ACM, 2013.

[16] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Computer & Communications Security–CCS*, pages 850–861. ACM, 2015.

[17] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Project Yellow Paper, 2014.

A Proof of security

A complete proof of security would duplicate most of the proof of semi-honest security of Path ORAM. We do not think reproducing that proof would be beneficial for the reader, so we instead make reference to arguments made in that proof where they are required, and we refer the reader to the original semi-honest proof [15] to see those details explained

We split the proof into two cases: the first is the case of a malicious server, and the second is the case of a malicious client.

Client-side security

In this case, we prove that for all real world adversarial servers $\mathcal{S}$ there exists an ideal world simulator $\mathsf{Sim}_{\mathcal{S}}$ such that for all environments $\mathcal{Z}$,

$$|\Pr[\textsc{Real}_{\hat{\mathcal{C}},\mathcal{S},\mathcal{Z}}(\lambda)=1]-\Pr[\textsc{Ideal}_{\hat{\mathcal{C}},\mathsf{Sim}_{\mathcal{S}},\mathcal{Z}}(\lambda)=1]| \leq negl(\lambda).$$

Before proving this result directly, we will display a hybrid in which all data and operations are dummy:

Lemma 1. *The view of the server $\mathcal{S}$ in the externally verifiable Path ORAM protocol with an honest client is computationally indistinguishable from the view of the server in Game 3 below, where the client performs dummy operations with dummy data.*

Proof. In order to prove the above, we will show a sequence of hybrids.

Game 0 Game 0 is the real world scenario.

Game 1 In Game 1 the client stores and updates the database locally. After each access, if the verifier does not output $\mathsf{cheat}_{\mathcal{C}}$ or $\mathsf{cheat}_{\mathcal{S}}$ (i.e., the access is successful) instead of decrypting the desired ciphertext the client reads the corresponding plaintext from its local database.

For the server to distinguish Game 0 and Game 1, the server must either give the client incorrect data and cause verification to succeed, or cause the output of the verifier to change compared to Game 0. The server cannot do the former, by the security of the hash function and Merkle tree. The server cannot do the latter, since the messages sent to the verifier do not change in this game. Therefore, Game 0 and Game 1 are indistinguishable.

Game 2 In Game 2, instead of uploading the real database to the server during setup, the client uploads a dummy database. The client still stores the real database locally. For each access, the client requests a read with the same index as what the client would request on the real database. While performing an access with the server, the client engages in the same verification procedure over the dummy ciphertexts. If verification of the dummy database fails, the client correspondingly aborts as if it was the real database.

First, note that the semi-honest ORAM scheme guarantees that reads are indistinguishable from writes. Thus, for the server to distinguish Game 1 from Game 2, the server must distinguish ciphertexts in Game 1 from Game 2, or distinguish hashes or signatures in Game 1 from Game 2, or see different behavior from the verifier.

By the CPA security of symmetric-key encryption, none of these things can happen. If they did, an adversary attacking the CPA security of the encryption scheme could simulate the entire interaction, using an encryption oracle to encrypt on the client's behalf, and use the server (or verifier) behavior to distinguish encryptions of real data from encryptions of dummy data. (This argument assumes that the signatures the client computes are computed with a different public key than the encryptions.)

Game 3 Game 3 is the same as Game 2, except that the client always reads dummy index $ind' = 0$ from the server.

The same argument given for the semi-honest security of Path ORAM implies that the server cannot distinguish the change in the client's access pattern as a result of the choices of path x on each step. Thus, we must only show that verification does not leak any access pattern. For Path ORAM, each path is verified in the same way; siblings are requested, and reused to recompute the new root hash. By the security of Merkle trees, if the server sends *any* hash incorrectly, the client notices: if the client was running an unverified access, a verified access is requested. If the client was running a verified access, then $\mathsf{cheat}_{\mathcal{S}}$ gets output by the verifier. Because the server already knows the client will (or will not) detect misbehavior, the client's actual detection and response adds no additional information.

□

In Game 3, the server $\mathcal{S}$ stores a dummy database which is always accessed with a dummy index. Given this hybrid, we construct an ideal world simulator $\mathsf{Sim}_{\mathcal{S}}$ that internally simulates $\mathcal{S}$, $\mathcal{V}$, and the modified client of Game 3. We assume, without loss of generality, that $\mathcal{S}$ sends its view to the environment $\mathcal{Z}$ during setup and each access. (Any other message sent to $\mathcal{Z}$ is necessarily a function of the view of $\mathcal{S}$, so it suffices to prove verifiability if $\mathcal{S}$ sends its view.)

Simulation Upon receiving a notification that the client has sent a setup request to $\mathcal{F}$ with a database of size $|D|$, $\mathsf{Sim}_{\mathcal{S}}$ internally runs the setup procedure for an instance of Game 3 as above with an honest client and server $\mathcal{S}$ with a dummy database of size $|D|$. If the internal client aborts, $\mathsf{Sim}_{\mathcal{S}}$ sends abort to $\mathcal{F}$; otherwise, $\mathsf{Sim}_{\mathcal{S}}$ sends ok to $\mathcal{F}$. Then, $\mathsf{Sim}_{\mathcal{S}}$ sends the view of $\mathcal{S}$ to $\mathcal{Z}$.

Upon receiving a notification that an access is occurring from $\mathcal{F}$, $\mathsf{Sim}_{\mathcal{S}}$ internally runs a dummy access $(\mathit{read}, 0, \perp)$ with $\mathcal{S}$. If Phase 1 aborts, $\mathsf{Sim}_{\mathcal{S}}$ sends $\mathit{vrfy}_{\mathcal{S}} = \mathsf{yes}$; otherwise, $\mathsf{Sim}_{\mathcal{S}}$ sends $\mathit{vrfy}_{\mathcal{S}} = \mathsf{no}$. If the access proceeds to Phase 2 and the verifier outputs $\mathsf{cheat}_{\mathcal{S}}$, $\mathsf{Sim}_{\mathcal{S}}$ sends $\mathit{fail}_{\mathcal{S}} = \mathsf{yes}$. If the verifier outputs $\mathsf{cheat}_{\mathcal{C}}$ or $\mathsf{success}$, $\mathsf{Sim}_{\mathcal{S}}$ sends $\mathit{fail}_{\mathcal{S}} = \mathsf{no}$. At the end of each access, $\mathsf{Sim}_{\mathcal{S}}$ sends the view of $\mathcal{S}$ to $\mathcal{Z}$.

We now need to show that the simulator $\mathsf{Sim}_{\mathcal{S}}$ as defined above, internally running Game 3, is such that the output bit of the environment is indistinguishable between the real world and ideal world. Lemma 1 already showed that the view of $\mathcal{S}$ (including the outputs of $\mathcal{V}$) is identical in Game 3 and the real world. Therefore $\mathsf{Sim}_{\mathcal{S}}$ outputs a view that is indistinguishable from the real-world view of $\mathcal{S}$. It is also clear from the construction of the simulator that client output is the same in both worlds. All that remains is to show that the output of $\mathcal{V}$ in the ideal world is indistinguishable from that in the real world.

Note that by sending the appropriate values of $\mathit{vrfy}_{\mathcal{S}}$ and $\mathit{fail}_{\mathcal{S}}$, $\mathsf{Sim}_{\mathcal{S}}$ can essentially choose the output of $\mathcal{V}$ in the ideal world, guaranteeing that it matches the output of the simulated verifier $\mathsf{Sim}_{\mathcal{S}}$ is running internally. The only exception to this is if the simulated verifier outputs $\mathsf{cheat}_{\mathcal{C}}$. We must show this happens with negligible probability. There are three cases in which this output might occur:

1. If $\sigma_{\mathcal{S}}$ does not verify correctly from the client.
2. If $\mathit{count}_{\mathcal{S}} \geq \mathit{count}_{\mathcal{C}} + 2$; i.e., if the root sent by the client is much too old, compared to the root sent by the server.
3. If the new signed root $\sigma'_{\mathcal{C}}$ does not verify or the new count $\mathit{count}'_{\mathcal{C}}$ is not one greater than $\mathit{count}_{\mathcal{C}}$.

The first and third cases cannot occur, since the client here is honest. To limit the second case, note that after each operation is complete, either one party has

been found cheating (in which case no additional operations will occur) or both parties now have signatures of matching counters. Therefore each access starts with matching counters. As discussed earlier, the server could move their counter one ahead of the client by not sending the final message in Phase 1, but the client will never sign a counter that has been increased by more than one during a single operation, so this cannot occur due to a signature actually received from the client. Therefore it must come from a forged signature, which happens with negligible probability.

Thus, the simulation is successful, and we have security against a malicious server.

Server-side security

In this case, we prove that for all real world adversarial clients $\mathcal{C}$ there exists an ideal world client $\mathsf{Sim}_{\mathcal{C}}$ such that for all server-side environments $\mathcal{Z}$,

$$|\Pr[\text{REAL}_{\mathcal{C},\hat{\mathcal{S}},\mathcal{Z}}(\lambda)=1] - \Pr[\text{IDEAL}_{\mathsf{Sim}_{\mathcal{C}},\hat{\mathcal{S}},\mathcal{Z}}(\lambda)=1]| \leq \mathit{negl}(\lambda).$$

As above, we may assume without loss of generality that $\mathcal{C}$ sends its view to $\mathcal{Z}$ each access, since any other message is a function of this view.

Simulation The ideal world simulator $\mathsf{Sim}_{\mathcal{C}}$ runs the real world protocol between client $\mathcal{C}$, an honest server, and the verifier internally, always sending the simulated view of $\mathcal{C}$ to $\mathcal{Z}$. When $\mathcal{C}$ runs the setup protocol successfully with the server using database D, $\mathsf{Sim}_{\mathcal{C}}$ sends D to $\mathcal{F}$.

Any time the simulated $\mathcal{C}$ carries out an operation in the internal simulation, $\mathsf{Sim}_{\mathcal{C}}$ carries out an operation in the ideal world, always requesting a read of index 0. If the simulated operation finishes without the involvement of the verifier, $\mathsf{Sim}_{\mathcal{C}}$ sends $\mathit{vrfy}_{\mathcal{C}} = \mathsf{no}$. Otherwise, $\mathit{vrfy}_{\mathcal{C}} = \mathsf{yes}$ and then $\mathit{fail}_{\mathcal{C}} = \mathsf{yes}$ only if the simulated verifier outputs $\mathsf{cheat}_{\mathcal{C}}$.

We now need to show that the environment cannot distinguish the real world from the ideal world with the above simulation. In the setup phase, the output of $\mathsf{Sim}_{\mathcal{C}}$ is identical to the view of $\mathcal{C}$, and the server will output notification of the setup and $|D|$ in both the real and ideal worlds if and only if $\mathcal{C}$ successfully concludes the setup protocol. Thus, we can assume the simulated server and $\mathcal{C}$ have successfully exchanged signed roots and counters. We then prove that a given access looks identical in both worlds.

It is clear that in each access the view of $\mathcal{C}$ in the real world matches the output of $\mathsf{Sim}_{\mathcal{C}}$ in the ideal world, since $\mathsf{Sim}_{\mathcal{C}}$ is simulated exactly the same real world interaction internally. It's also clear that the output of the server is the same in both worlds, since whenever the real world server notifies $\mathcal{Z}$ that an access is occurring, $\mathsf{Sim}_{\mathcal{C}}$ sends a message to $\mathcal{F}$ that causes the same thing to happen in the ideal world. That means the only difference between worlds can come from the output of the verifier.

Whenever the verifier would output $\mathsf{cheat}_{\mathcal{C}}$ or $\mathsf{success}$ in the real world, $\mathsf{Sim}_{\mathcal{C}}$ sends $\mathit{vrfy}_{\mathcal{C}}$ and $\mathit{fail}_{\mathcal{C}}$ values that cause the same thing to happen in the ideal world. So we must show only that an output of $\mathsf{cheat}_{\mathcal{S}}$ occurs with negligible probability in the real world (or, equivalently, in the internal simulation of $\mathsf{Sim}_{\mathcal{C}}$). We consider each case where the verifier might decide on such an output:

1. If σ_C on any root sent by the server does not verify correctly.
2. If $\mathit{count}_{\mathcal{S}}$ is incorrect; i.e., if $\mathit{count}_{\mathcal{S}} = \mathit{count}_{\mathcal{C}} + 1$ or $\mathit{count}_{\mathcal{S}} < \mathit{count}_{\mathcal{C}}$.
3. If the Merkle proof for data P does not match the agreed upon root $\mathit{root}_{\mathcal{C}}$.
4. If the new signed root $\sigma'_{\mathcal{S}}$ from the server is not the correct value or if the new signature $\sigma_{\mathcal{S}}$ does not verify correctly.

All cases but the second are impossible. The honest server will only send σ_C values that it received from the client (and that verified correctly when received). The Merkle proof and new counter and signature values will also always be correct. The second case requires a more nuanced examination. If the $\mathit{count}_{\mathcal{C}}$ value received from the client (through the verifier) is one less than the most recent count the server has seen, the server will roll back the database to match that count value, meaning that $\mathit{count}_{\mathcal{S}} = \mathit{count}_{\mathcal{C}} + 1$ can never occur. Furthermore, the server never signs a given count value until after it has seen the same value signed by the client. As a result, $\mathit{count}_{\mathcal{S}} < \mathit{count}_{\mathcal{C}}$ cannot occur unless the client has forged a signature, which happens with negligible probability.

This completes the proof of security against a malicious client, and therefore the protocol is a secure externally verified ORAM protocol.

B Malicious-Secure Ring ORAM

Here we prove that the Ring ORAM variant from Section 5.2 achieves malicious security. We borrow the

simulation-based definition of secure ORAM with a malicious server from [4]. This definition, in contrast with the definition of externally verifiable ORAM, considers only client-side security, so only the server is allowed to arbitrarily deviate from the protocol.

Ideal world Here, $\mathcal{F}$ is an ideal functionality that locally stores the database, which interfaces with the client $\mathcal{C}$ and server $\mathcal{S}$.

Setup An environment $\mathcal{Z}$ gives a database D to $\mathcal{C}$, who forwards D to $\mathcal{F}$. Then, $\mathcal{F}$ tells $\mathcal{S}$ the size of the database $|D|$. Then, $\mathcal{S}$ gives ok or abort to $\mathcal{F}$, who forwards ok or $\perp$ to $\mathcal{C}$ accordingly. If $\mathcal{C}$ receives $\perp$, the execution ends.

Access Each time step, the environment $\mathcal{Z}$ gives $\mathcal{C}$ a command $(op, ind, data)$, and $\mathcal{C}$ forwards it to $\mathcal{F}$. $\mathcal{F}$ then notifies $\mathcal{S}$ that an operation is happening. In response, $\mathcal{S}$ sends either ok or abort to $\mathcal{F}$. If $\mathcal{S}$ gave ok, $\mathcal{F}$ fulfills the request from $\mathcal{C}$ and sends $\mathcal{C}$ any requested data. If $\mathcal{S}$ gave abort, $\mathcal{F}$ gives $\perp$ to $\mathcal{C}$. $\mathcal{C}$ then forwards this data to $\mathcal{Z}$.

After the setup procedure and each access, $\mathcal{S}$ may send a message to $\mathcal{Z}$.

Once all $\text{poly}(\lambda)$ accesses have been completed, $\mathcal{Z}$ outputs a bit. Define the random variable $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda)$ to be this output.

Real world In the real world, the environment $\mathcal{Z}$ gives $\mathcal{C}$ a database D. $\mathcal{C}$ runs the setup procedure with the real world server $\mathcal{S}$. If the setup protocol aborts, the execution ends. At each time step, $\mathcal{Z}$ gives $\mathcal{C}$ the tuple $(op, ind, data)$, who runs the corresponding access with $\mathcal{S}$. The client then forwards any data received, or $\perp$ if the protocol aborted, to $\mathcal{Z}$. After the setup procedure and each access, $\mathcal{S}$ may send a message to $\mathcal{Z}$.

After all access have been completed, $\mathcal{Z}$ outputs a bit. Define the random variable $\textsc{Real}_{\Pi,\mathcal{S},\mathcal{Z}}(\lambda)$ to be the final bit output by $\mathcal{Z}$ in the real world scenario.

Definition 2 A protocol Π is *malicious-secure* if for all real world servers $\mathcal{S}$, there exists an ideal world simulator $\mathsf{Sim}_{\mathcal{S}}$ such that for all environments $\mathcal{Z}$, there exists a negligible function *negl* such that

$$|\Pr[\textsc{Real}_{\Pi,\mathcal{S},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F},\mathsf{Sim}_{\mathcal{S}},\mathcal{Z}}(\lambda) = 1]| \leq negl(\lambda).$$

♢

Having defined security, we now proceed with the proof. We give a simulator and then use a hybrid argument to show that the real and ideal worlds are indistinguishable.

The simulator The simulator is analogous to the above one for Path ORAM, but for a real world instance of Ring ORAM using dummy data.

The simulator $\mathsf{Sim}_{\mathcal{S}}$ runs the real world protocol locally between client $\mathcal{C}$ and server $\mathcal{S}$. When the functionality $\mathcal{F}$ notifies $\mathsf{Sim}_{\mathcal{S}}$ that a setup operation is occurring with database size $|D|$, $\mathsf{Sim}_{\mathcal{S}}$ runs the real world setup protocol between $\mathcal{C}$ and $\mathcal{S}$ with a dummy database of size $|D|$. If the setup protocol aborts, $\mathsf{Sim}_{\mathcal{S}}$ sends abort to $\mathcal{F}$; otherwise, $\mathsf{Sim}_{\mathcal{S}}$ sends ok to $\mathcal{F}$.

When $\mathcal{F}$ notifies $\mathsf{Sim}_{\mathcal{S}}$ that an access is occurring, $\mathsf{Sim}_{\mathcal{S}}$ runs an access between $\mathcal{C}$ and $\mathcal{S}$ with dummy index $ind' = 0$. This access includes a ReadPath operation and any EvictPath or EarlyReshuffle operations that will are required after the ReadPath has been performed. If the client aborts, the simulator sends abort to $\mathcal{F}$. Otherwise, it sends ok.

Game 0 Game 0 is the real world scenario.

Game 1 In Game 1, the client stores the database locally, including metadata. If verification succeeds after each access, instead of decrypting the desired ciphertext or metadata, the client reads the corresponding plaintext from its local database.

To distinguish Game 0 and Game 1, the server must give the client incorrect data (or metadata) and cause verification to succeed. The server is unable to do so, by the security of authenticated encryption and Merkle trees. (The client stores the updated freshness counter and knows the position index, so blocks cannot be rearranged. The server would have to forge a valid encryption or find a collision in the hash underlying the Merkle tree.) Therefore, Game 0 and Game 1 are indistinguishable.

Game 2 In Game 2, instead of uploading the real database to the server, the client uploads a dummy database. The client still stores the real database locally. For each access, the client requests a read with the same index as what the client would request on the real database. After completing an access with the server, the client engages in the same verification procedure; that is, the client verifies the dummy database.

First, note that reads are indistinguishable from writes by the same argument that applies in the semi-honest ORAM case. By the CPA security of symmetric-key encryption, the server cannot tell dummy data from real data. Since all Merkle tree hashes are functions only of ciphertexts, they cannot add any more information. Thus, Game 1 and Game 2 are indistinguishable.

Game 3 Game 3 is the same as Game 2, except that the client always accesses dummy index $ind' = 0$ from the server.

Again, the argument from the semi-honest case applies, unless the verification steps give the server additional information. However, the client verifies successful decryption of every block read every access, and any change by the server will result in an abort (unless an encryption was forged) so the server knows ahead of time how the client will behave. Thus, no information new information is gained from this behavior, and Game 2 and Game 3 are indistinguishable.

Game 3 is equivalent to the ideal world, so the present construction is malicious secure.

XOR Technique Using the XOR technique, instead of sending $\log(n)$ blocks for every ReadPath access, the server sends the XOR of all of these blocks. All but one of these blocks are dummy blocks, so the client may recompute the desired encrypted blocks. The client may then abort as usual if the recomputed block does not decrypt successfully.

Game 0 is still indistinguishable from Game 1, since this does not enable the server to forge any encryption it couldn't before; any computation that the server could do here, the server could have done in the original ReadPath protocol. Also, Game 1 is still visibly indistinguishable from Game 2.

To show that Game 2 is indistinguishable from Game 3, we need to show that the decision for the client to abort during ReadPath using XOR does not leak any more information than during ReadPath without XOR. First, note that once a block is involved in a ReadPath operation, that block is now invalidated; thus, we do not need to consider multiple ReadPath operations that involve overlapping sets of blocks.

If the server alters an odd number of blocks at bit position i during a ReadPath operation, then the corresponding bit will be flipped, and corresponding decryption will fail. Similarly, if the server alters an even number of blocks at bit position i, then the decryption will succeed. Thus, whether or not the client aborts is purely a function of how many bits are flipped at each position. This means that the server already knows whether the client will abort, meaning the server can infer nothing about the access pattern from the client's behavior.

Thus, the server cannot gain any information using the XOR technique, so Game 2 is indistinguishable from Game 3. Using the same simulator, we see that the construction is malicious secure with the XOR technique.

C Detailed Description of Externally Verifiable Ring ORAM

Below are detailed descriptions of the honest execution of Phase 1 for each operation of Ring ORAM. We use D_B to represent the data stored in bucket B, and we use $count_O$ and $count_M$ to represent the counters associated with the roots of OT and MT respectively.

ReadPath During ReadPath, the client downloads, modifies, and re-uploads metadata along a path. The client then requests specific block offsets within each bucket along a path. The server sends the client the relevant data blocks, potentially XORed together. Data blocks received during this operation are verified by checking successful decryption.

Note that the metadata tree is modified, while the ORAM tree is not. Thus, the counter $count_M$ for the metadata tree may be greater than the counter $count_O$ for the ORAM tree.

1. Client requests metadata path P_M, and server responds with P_M and Merkle proof MP_M.
2. Using previously stored root r_M, if $\text{ReconstructRoot}(P_M, MP_M) \neq r_M$, client aborts.
3. Client runs ReadPath, and requests blocks B_i from server. During this process, client modifies metadata to obtain P'_M, according to the Ring ORAM protocol.
4. If using the XOR technique, client receives an XORed path of blocks B^*. Client reconstructs block B, by using reconstructed dummy blocks. Otherwise, client receives a path of blocks B_i.
5. Client verifies each B_i (or B) using position indices, stored freshness counters, and by checking successful decryption. If any block fails to verify, the client aborts.
6. Client constructs new metadata root r'_M from P'_M and MP_M.
7. Client sends server signed metadata root $(r'_M, count'_M, \mathsf{Sign}_{\mathcal{C}}(r'_M || count'_M))$, with updated counter $count'_M = count_M + 1$. Server responds with $(r'_M, count'_M, \mathsf{Sign}_{\mathcal{S}}(r'_M || count'_M))$. If the server's signature or counter is incorrect, the client aborts. If the client's signature or counter is incorrect, the server aborts, which causes the client to abort.

EvictPath During EvictPath, the client calls ReadBucket along an entire path of buckets. Then, the client

calls WriteBucket along that same path. Metadata is operated on throughout.

1. Client requests metadata path P_M, and server responds with P_M and Merkle proof MP_M. (The selection of path is specified by the Ring ORAM protocol.)
2. Using previously stored root r_M, if ReconstructRoot$(P_M, MP_M) \neq r_M$, client aborts.
3. For each bucket in path P (root to leaf),
   (a) Using metadata, client requests Z blocks from bucket B. Server sends back the requested Z blocks $\{b_j\}$, and S hashes $\{h_k\}$ of the other blocks.
   (b) Client forms hashes of received blocks $\{b_j\}$, and combines these hashes with received hashes $\{h_k\}$ to form a collection of $Z+S$ block hashes D_B.
   (c) Client uses leaf hashes D_B to construct bucket root r_B.
4. Server sends client Merkle proof MP_O for the ORAM tree.
5. Using previously stored root r_O and bucket roots $\{r_B\}$ from above, if ReconstructRoot$(\{r_B\}, MP_O) \neq r_O$, client aborts.
6. For each bucket B in path P (leaf to root),
   (a) Client sends $Z+S$ blocks to server in bucket B. Client hashes these blocks to form D'_B.
   (b) Client uses leaf hashes D'_B to construct bucket root r'_B.
7. Client reconstructs new ORAM tree root r'_O using MP_O and the bucket roots $\{r'_B\}$.
8. Client sends server signed ORAM tree root $(r'_O, count'_O, \mathsf{Sign}_{\mathcal{C}}(r'_O||count'_O))$, with updated counter $count'_O = count_O + 1$. Server responds with $(r'_O, count'_O, \mathsf{Sign}_{\mathcal{S}}(r'_O||count'_O))$. If the server's signature or counter is incorrect, the client aborts. If the client's signature or counter is incorrect, the server aborts, which causes the client to abort.
9. Throughout the above steps, metadata is modified to eventually obtain P'_M. Client reconstructs new metadata tree root r'_M using P'_M and MP_M.
10. Client sends server signed metadata root $(r'_M, count'_M, \mathsf{Sign}_{\mathcal{C}}(r'_M||count'_M))$, with updated counter $count'_M = count_M + 1$. Server responds with $(r'_M, count'_M, \mathsf{Sign}_{\mathcal{S}}(r'_M||count'_M))$. If the server's signature or counter is incorrect, the client aborts. If the client's signature or counter is incorrect, the server aborts, which causes the client to abort.

EarlyReshuffle During EarlyReshuffle, the client calls ReadBucket and WriteBucket on select buckets along a path. This operation is similar to EvictPath, but not all buckets along the path will be updated. Additionally, in EvictPath the operations were batched such that all ReadBucket operations happened before all WriteBucket operations. Here, ReadBucket and WriteBucket operations alternate.

1. Client requests metadata path P_M, and server responds with P_M and Merkle proof MP_M. The selection of path is specified by the Ring ORAM protocol.
2. Using previously stored root r_M, if ReconstructRoot$(P_M, MP_M) \neq r_M$, client aborts.
3. Along the entire path, server sends collection of bucket roots $\{r_B\}$, with Merkle proof MP_O.
4. Using previously stored root r_O, if ReconstructRoot$(\{r_B\}, MP_O) \neq r_O$, client aborts.
5. Using metadata from P_M, client constructs list L of what buckets need to be updated according to the Ring ORAM protocol. Client sends list L to server.
6. For each bucket B in L (ordered from root to leaf):
   (a) Client calls ReadBucket and requests Z blocks from bucket B. Server sends back the requested Z blocks $\{b_j\}$, and S hashes $\{h_k\}$ of the other blocks.
   (b) Client forms hashes of received blocks $\{b_j\}$, and combines these hashes with received hashes $\{h_k\}$ to form a collection of $Z+S$ block hashes D_B.
   (c) Client uses leaf hashes D_B to reconstruct bucket root r^*_B. If $r^*_B \neq r_B$ from above, client aborts.
   (d) Client modifies metadata in B and sends $Z+S$ blocks to bucket B. Client hashes these blocks to form D'_B. (Client also modifies metadata in P_M accordingly.)
   (e) Client uses leaf hashes D'_B to construct bucket root r'_B.
7. For each bucket B not in L, let $r'_B = r_B$.
8. Client constructs new ORAM tree root r'_O, using $\{r'_B\}$ and MP_O.
9. Client sends server signed ORAM tree root $(r'_O, count'_O, \mathsf{Sign}_{\mathcal{C}}(r'_O||count'_O))$, with updated counter $count'_O = count_O + 1$. Server responds with $(r'_O, count'_O, \mathsf{Sign}_{\mathcal{S}}(r'_O||count'_O))$. If the server's signature or counter is incorrect, the client aborts. If the client's signature or counter is incorrect, the server aborts, which causes the client to abort.
10. Throughout the above steps, metadata is modified to eventually obtain P'_M. Client reconstructs new metadata tree root r'_M using P'_M and MP_M.

11. Client sends server signed metadata root $(r'_M, count'_M, \mathsf{Sign}_{\mathcal{C}}(r'_M||count'_M))$, with updated counter $count'_M = count_M + 1$. Server responds with $(r'_M, count'_M, \mathsf{Sign}_{\mathcal{S}}(r'_M||count'_M))$. If the server's signature or counter is incorrect, the client aborts. If the client's signature or counter is incorrect, the server aborts, which causes the client to abort.

D Proof of Security for Externally Verifiable Ring ORAM

This proof largely follows the same argument as was used for the externally verifiable version of Path ORAM. We begin with a lemma that captures the part of the proof with the most additional complexity, and then use this lemma to proceed through the proof as we did previously.

Lemma 2. *If an operation ends successfully (i.e., in Phase 1 or in Phase 2 with the verifier outputting* success*), with all but negligible probability any data received by the client is correct and the client's updated Merkle tree roots accurately reflect a tree where new data has been written as expected.*

Proof. This is immediate for the metadata tree, since it is an augmented Merkle tree interfaced with in the same manner as in externally verifiable Path ORAM.

Recall that all data blocks are encrypted using authenticated encryption. During ReadPath, data blocks are verified by checking successful decryption. For the client to not abort while reading block B_i, decryption must succeed with the correct freshness counter and position index; i.e., the block must be authentic, up-to-date, and in the correct position in the ORAM tree. The server cannot forge this encryption with greater than negligible probability.

During EvictPath, the client uses the Merkle tree to authenticate data. If this does not cause the client to abort, then with all but negligible probability each received bucket root r_B is consistent with the stored ORAM tree root r_O, by the security of Merkle trees. In turn, with all but negligible probability each bucket root r_B is consistent with the received blocks and block hashes in that bucket. If the received data and Merkle proof is correct, then it follows that the client's updated root is correct for the newly modified database.

EarlyReshuffle is similar, but the server also provides the client with bucket roots r_B not included in the list L of buckets to request. If the Merkle proof for the ORAM tree succeeds, these are similarly consistent with the stored root r_O with all but negligible probability.

Thus, if the Merkle proof succeeds, all received blocks, block hashes, and bucket roots are confirmed to be correct with all but negligible probability, and all updates will then occur correctly. □

Client-side security The proof for externally verifiable Ring ORAM is essentially the same as for externally verifiable Path ORAM. We use the above two lemmas to show that the view of a server is indistinguishable between the real world and scenario where all operations are dummy:

Lemma 3. *The view of the server $\mathcal{S}$ in the externally verifiable Ring ORAM protocol with an honest client is computationally indistinguishable from the view of the server in Game 3 below, where the client performs dummy operations with dummy data.*

Proof. As before, we use a hybrid argument.

Game 0 Game 0 is the real world scenario.

Game 1 In Game 1 the client stores and updates the database locally. After each access, if the verifier does not output $\mathsf{cheat}_{\mathcal{C}}$ or $\mathsf{cheat}_{\mathcal{S}}$ (i.e., the access is successful) instead of decrypting the desired ciphertext the client reads the corresponding plaintext from its local database.

For the server to distinguish Game 0 and Game 1, the server must either give the client incorrect data and cause verification to succeed, or cause the output of the verifier to change. The server cannot do the former; by Lemma 2, if the client does not abort from Phase 1, the client must have the correct data with all but negligible probability. By the security of Merkle trees, if the verifier does not output $\mathsf{cheat}_{\mathcal{C}}$ or $\mathsf{cheat}_{\mathcal{S}}$ during Phase 2, the verifier must have given the client the correct data. The server cannot do the latter, since the interaction with the verifier does not change in this game.

Therefore, Game 0 and Game 1 are indistinguishable.

Game 2 In Game 2, instead of uploading the real database to the server, the client uploads a dummy database. The client still stores the real database locally. For each access, the client requests a read with the same index as what the client would request on the real database. While performing an access with the

server, the client engages in the same verification procedure over the dummy ciphertexts. If verification of the dummy database fails, the client correspondingly aborts as if it was the real database.

First, note that the standard argument from the proof of security for semi-honest (or malicious) Ring ORAM that reads are indistinguishable from writes still applies, with minor revision. If ReadPath is the only operation called, the only additional information exchanges are Merkle tree roots and their signatures, which reveal no information. If EvictPath or EarlyReshuffle is run, the only additional information sent (compared to the malicious case) is the Merkle proof in tree OT (essentially hashes of other blocks/buckets), which again reveals no additional information.

Thus, for the server to distinguish Game 1 from Game 2, the server must distinguish ciphertexts in Game 1 from Game 2, or distinguish hashes or signatures in Game 1 from Game 2, or see different behavior from the verifier. The same argument in Game 2 of Lemma 1 applies to show this is impossible and therefore that Game 1 and Game 2 are indistinguishable.

Game 3 Game 3 is the same as Game 2, except that the client always reads dummy index $ind' = 0$ from the server.

Again, the argument from the semi-honest Ring ORAM security proof applies here. The same additional information is added here as above (namely, using authenticated encryption, signed roots of Merkle trees, and Merkle proofs), and this information is all a function only of the ciphertexts, which the server can already see. It follows that Game 2 is indistinguishable from Game 3.

□

Given the above hybrid, we use the same simulator as in Appendix A, but running the above Game 3 instead. The Ring ORAM verifier behaves essentially the same as the Path ORAM verifier, since both trees are being verified simultaneously. Because of this, the same argument from Appendix A holds here as well.

Server-side security The same simulator and basic argument as in the proof for server-side verifiability for Path ORAM in Appendix A also applies here. Recall that in the proof for Path ORAM, the simulator $\mathsf{Sim}_{\mathcal{C}}$ internally runs a real world instance of the ORAM protocol between $\mathcal{C}$, an honest server, and the verifier; then, $\mathsf{Sim}_{\mathcal{C}}$ outputs the correct messages to the ideal functionality in order to imitate the same behavior with respect to the output of the verifier. For this proof, $\mathsf{Sim}_{\mathcal{C}}$ is the same as in Appendix A, but running a real world instance of externally verifiable Ring ORAM.

Specifically, $\mathsf{Sim}_{\mathcal{C}}$ outputs $vrfy_{\mathcal{C}}$ and $fail_{\mathcal{C}}$ in order to correspond with the internally-simulated verifier's output of $\mathsf{success}$ or $\mathsf{cheat}_{\mathcal{C}}$. The result follows, then, as long as $\mathcal{C}$ cannot force the verifier to output $\mathsf{cheat}_{\mathcal{S}}$ with non-negligible probability. The same analysis holds as in Appendix A to show that this is impossible. The only modification is that the argument must be applied separately to OT and MT.