ConcurORAM: Multi-client concurrency for RingORAM

ABSTRACT

Oblivious RAM (ORAM) technology has advanced rapidly in recent years as an increasing amount of data is outsourced to remote storage. Although tree based ORAMs such as PathORAM [18] and RingORAM [15] have achieved near-optimal bandwidth blowup for *single client* scenarios, their low overall throughput due to high latency of access makes them prohibitive in multi-client scenarios.

In this paper, we propose ConcurORAM, a multi-client concurrent variant of RingORAM that reduces waiting for concurrent client accesses and increases overall throughput. ConcurORAM leverages the asynchronous nature of RingORAM queries to support concurrent client accesses until the next round of eviction. Further to support concurrent accesses, the position map for RingORAM is stored in a pyramid ORAM using the PD-ORAM deamortization abstraction detailed in [21].

ConcurORAM has been implemented and tested. It shows an overall increase in throughput by a factor of **[Anrin: to be filled]** and a reduction in query execution time by a factor of **[Anrin: to be filled]** over a standard implementation of RingORAM for a multi-client scenario.

1. INTRODUCTION

With an increasing amount of confidential data being outsourced to remote storage, ensuring the privacy of this data is critical. As demonstrated in previous work [11], simply encrypting the data is not enough. Even on encrypted data, the sequence of locations read and written to the storage can leak information regarding the user's *access pattern* and the data stored.

Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to hide its data access patterns from an untrusted storage hosting the data. Informally, the ORAM adversarial model prevents an adversary from distinguishing between multiple equal length sequence of queries made by the client to the server.

Since the original ORAM construction by Goldreich and Ostroevsky [6], a large volume of previous literature [8, 15, 16, 18, 20, 21] has been dedicated to developing more efficient ORAM constructions. PathORAM [18], based on the original binary tree ORAM construction by Shi *et al.* [16] is widely accepted to be asymptotically the most *bandwidth efficient* ORAM. RingORAM [15] further optimizes PathORAM [18] for practical deployment by reducing the constants in the asymptotic bandwidth for PathORAM. Even for ORAMs that divide the data into multiple sub-ORAMs such as ObliviStore [17] and CURIOUS [2], [2] shows that PathORAM [18] is the most suitable ORAM for sub-ORAM design in terms of cost and bandwidth.

Although, recent tree based ORAM designs have achieved near-optimal *bandwidth* for single client scenarios, one critical challenge, yet to be addresses, is to make these ORAMs multi-client compatible for concurrent non-overlapping access (access for different data items). As a motivating example, consider an enterprise that outsources confidential data to a remote storage that deploys an ORAM, and provides access to a group of employees (users). These users should be able to perform non-overlapping queries without a significant performance overhead in comparison to a scenario where there is only one user accessing the ORAM.

Note that it is trivial to deploy a standard tree based ORAM without concurrency to support multiple clients by sharing the key to the ORAM and storing all related datastructures (the stash and the position map) on the storage server to ensure consistency of state. In this case, only *one* client can access the position map, the stash and the tree at one time while the other concurrent clients must wait for this client to finish. This reduces the overall throughput and increases the query response time by a factor of the number of concurrent clients. A client (in the worst case) might need to wait for *all* other clients to finish before retrieving the required data item. Since, ORAMs have high latency of access (due to multiple round trips of $O(logN)$ data items), this implies that a client would need to wait a significant amount of time before being able to proceed with the query.

In this paper, we propose ConcurORAM , a mechanism to support multi-client concurrency for tree-based ORAMs without sacrificing security. Our work is based on the concurrency scheme proposed by Williams *et al.* [21]. However, as we detail below there are significant challenges to directly adapting the techniques proposed in [21] for tree based ORAMs.

Concurrency for position map. Tree based ORAMs use a position map to store mappings from the logical IDs of data items to the leaf IDs in the tree they are mapped to. Specifically, a data item mapped to leaf ID l can reside in any of the nodes along the path from the root to leaf l. In a single client scenario, the position map can be stored at the client side. For a multi-client scenario the position map must be stored on the server to ensure consistency among the clients. As introduced in [16], the position map can be stored at the server (to reduce client side storage) by dividing the position map into fixed sizes blocks and storing them in recursively smaller ORAMs. Thus, an access in this case, requires reading the position map from the smaller ORAMs to obtain the leaf ID for the required data item and then reading the corresponding path to retrieve the data item. Since, the position map is stored recursively, an ORAM storing the position map also has a position map. To ensure that each successive position map is smaller (to ensure that the recursion terminates with an ORAM that has a a constant size position map), each block in the ORAMs must store multiple position map entries.

However, this technique presents a security problem in a multi-client scenario. To illustrate, consider two clients that want to access different data items whose position map entries are stored in the same position map block. In this case, concurrent access by the clients will allow the server to correlate the two client accesses which would leak access pattern privacy in the multi-client scenario.

As one of the main insights, ConcurORAM stores the entries of the position map in a pyramid ORAM ([6, 21]) with multiple levels that uses a hash function to map position map entries to buckets in a level. Note that since the location of an entry is randomized (due to the uniform hash function used), concurrent clients accessing the same bucket, does not leak any correlation between the items queried by the clients. Specifically, ConcurORAM use the concurrent version of PD-ORAM as used in PrivateFS [21] to ensure concurrency for queries.

Asynchronous operations. Tree based ORAMs divide accesses into two parts – fetching data (reading a root to leaf path) and eviction (writing back the read data to a root to leaf path). ORAMs that couple eviction with queries (such as PathORAM [18] and CLP-ORAM [3]) must be accessed synchronously to ensure consistency in multi-client scenarios. Thus, to support concurrent queries the fetching and eviction must be decoupled. Fortunately, RingORAM [15] provides a mechanism to support multiple fetches before an eviction. ConcurORAM uses RingORAM and allows a fixed number of concurrent queries followed by an eviction by a single client.

ConcurORAM has been implementd and shows an increase in throughput by a factor of **[Anrin: This number to be filled]** over a standard implementation of RingORAM [15] used non-concurrently for multiple clients.

2. RELATED

ORAMs have been well-researched since the seminal work by Goldreich and Ostroevsky [6]. ORAM constructions in literature can be broadly classified into two categories: pyramid based constructions and tree based constructions.

2.1 Pyramid based ORAM

A pyramid based construction organized the data in $logN$ levels. Level i consists of 4^i data blocks assigned to one of 4^i buckets as determined by a secure hash function. Due to hash collisions, each bucket can contain up to $O(lnN)$ blocks.

The first pyramid based construction was provided by Goldreich and Ostroevsky [6] which achieves an amortized communication complexity of $O(log^3 N)$ and requires only logarithmic storage at the client side.

Read. To obtain a particular block, a client scans *one* bucket from each level as determined by the hash of the logical block ID of that block. The ORAM maintains two invariants: i) a client access never reveals the level at which a block has been found, ii) a block is accessed from a particular location only once. To ensure (i), after a client has found the required block at a particular level, the client continues scanning a *random* bucket from all the levels below it. Once all the levels have been queried, the client reencrypts the accessed block and places it in the top level. This maintains (ii) since it ensures that a block that has been accessed earlier will be located in the top level for the next query for the same block. The rest of the search pattern will be random.

Write. Writes proceed in exactly the same way as the reads with the exception that the updated value of the accessed block is placed in the top level. The same data access pattern for both reads and writes and semantic encryption ensure that the server cannot learn the purpose (read or write) of an access.

Overflow. The top level overflows after a fixed number of accesses since queried blocks are invariably placed there. On an overflow, the top level is merged with the second level and reshuffled using a new secure hash function. The reshuffling is done obliviously using a sorting network [6, 7, 20]. Each level overflows once after the level above has overflown twice. To ensure invariant (i), all buckets contain the same number of blocks (either real or dummy) after a reshuffle.

The most expensive step of the pyramid ORAM is the reshuffle. Various mechanisms have been proposed to make the reshuffle more efficient. Williams and Sion [1] show how to achieve an amortized construction with $O(log^2 N)$ communication complexity under $O(\sqrt{N})$ client storage using an oblivious merge sort. Williams *et al.* further use this mechanism to build an ORAM with $O(logN)$ access complexity and $O(log^2 N)$ overall communication complexity by storing an encrypted bloom filter on the server and retrieving one block from a level.

Pinkas *et al.* [14] use *cuckoo hashing* and randomized shell sort [7] over the original Goldreich and Ostroevsky solution [6] and achieve an amortized communication complexity of $O(log^2 N)$ with constant client side storage. However, Goodrich *et al.* [9] highlight a leak in the construction in [14] and provide an alternate construction thats achieves an amortized communication complexity of $O(logN)$ under the assumption of $O(N^{1/r})$ client storage with $r > 1$.

Although, the above mentioned constructions improve upon the original solution in [6], client queries still need to wait for the duration of a reshuffle. De-amortized constructions allow queries and reshuffles to proceed together and thus eliminate clients waiting for reshuffles after a level overflow. Goodrich *et al.* [10] show how to de-amortize the original square root solution and hierarchical solution [6] and achieve a worst-case complexity of $O(logN))$ in the presence of $O(n^r)$ client side storage where $r > 0$. In [12], Kushilevitz *et al.* use cuckoo hashing and rotating buffers to provide a de-amortized construction of the original hierarchical solution [6] which achieves a worst-case communication complexity of $O(log^2N/loglogN)$.

PD-ORAM: Unlike the de-amortization techniques used in [10, 12] where each query performs an additional fixed amount of work for the reshuffle, the PD-ORAM [21] de-amortization abstraction performs a reshuffle in the background while monitoring progress to ensure that a level is ready after a reshuffle as soon as it is required. Further, queries can proceed simultaneously through a read-only variant of the level while the reshuffle takes place and ensures roughly similar query costs.

2.2 Tree-based ORAM

In contrast to de-amortized ORAM constructions, tree-based ORAMs are naturally un-amortized (the worst-case query cost is equal to the average cost). A tree based ORAM organizes the data as a binary (or ternary) tree. Each node of the tree is a bucket which can contain multiple blocks. A block is randomly mapped to a leaf in the tree. The ORAM maintains the following invariant: a block resides in any one of the buckets on the path from the root to the leaf to which the block is mapped. The position map which maps blocks to leaves is either stored on the client ($O(N)$ storage required) or recursively on the server for $O(1)$ client side storage at the cost of $O(log^2N)$ increase in communication complexity.

To access a particular block, the client downloads all the buckets (or one element from each bucket) along the path from the root to the leaf to which the block is mapped. Once the block has been read, it is remapped to a new leaf and *evicted* back to the tree. Various eviction procedures have been proposed in literature [3, 15, 16, 18, 19]

Binary Tree ORAM: The original tree-based ORAM proposed by Shi *et al.* [16] places the accessed block at the root of the tree and evicts a constant number of blocks from nodes at a particular depth to children nodes. In this case, the buckets need to be at least sized $O(logN)$ and the overall access complexity for the construction is $O(log^3N)$.

PathORAM: In PathORAM [18], eviction takes place by writing back the same path that was read and placing the remapped block along the path at a node that intersects with the path to the new leaf to which the block is remapped. Further, PathORAM [18] uses constant sized blocks in the presence of a logarithmic-sized client side *stash* to handle overflows.

RingORAM: RingORAM decouples fetching a path during an access and eviction, by evicting along a deterministically chosen path after a fixed number of fetches. Similar to [5], the path is chosen in the reverse-lexicographical order for better eviction. Further, RingORAM uses larger buckets but reads only one block per bucket during a query as determined by a per-bucket metadata. The required block is read from the bucket that contains it while dummy blocks are read from the rest of the buckets on the path.

2.3 Recent Developments

Recent work [2] has shown that for practical deployment on clouds, ORAMs such as ObliviStore [17] and CURIOUS [2] that divide the data into constant sized sub-ORAMs perform better than both pyramid ORAMs and tree based ORAMs. CURIOUS uses PathORAM as the primitive for the sub-ORAM construction. Alternative ORAM constructions [4, 13] achieve $O(1)$ communication complexity at the cost of server side computation using homomorphic encryption. However, due to expensive homomorphic computations, these ORAMs are not yet practical for deployment.

3. MODEL

Deployment. ConcurORAM considers a deployment model with two parties: the ORAM clients (with limited local storage) and the ORAM server (a remote storage that hosts the clients' data). The server stores data as fixed sized "blocks". ConcurORAM considers N blocks of outsourced data on the server. Clients also access data in blocks addressed by a logical block ID denoted by id. The logical address space for all blocks is shared by the clients. The parties engage in an interactive query-response based protocol established by ConcurORAM . The communication channel between the clients and the server is considered secure using SSL.

Clients. Clients are considered honest in the ConcurORAM model and do not interact with each other. Further, the clients share the key to the ORAMs and the secret hash functions used for PD-ORAM, which are stored encrypted on the server. Clients can engage the server without having any knowledge of other client states. Any locking mechanism (as required by the protocol) is imposed by the server. ConcurORAM does not consider the case of malicious clients.

Server. ConcurORAM considers an untrusted server that is honest but curious and does not deviate from the ConcurORAM protocol. The server can observe all requests and try to correlate them by saving and comparing snapshots (state of the ORAMs after each query). It stores the ORAM keys, hash functions and other access counters as required by the ConcurORAM protocol. Further, the server maintains and duly increments the counters. ConcurORAM does not consider a malicious server than can mount replay attacks and "fork" client views.

Security challenge. Any system that supports multi-client concurrent access for ORAMs needs to prevent two possible security leaks -

- Correlation between data items concurrently accessed in a single round.
- Correlation between data items accessed in successive rounds of one or more concurrent accesses.

In the above context, we define multi-client concurrent access security for ORAMs as a security game, where the challenger is a fixed set of clients, $\mathcal{C} = \{c_1, c_2, \dots c_n\}$, and the adversary, $\mathcal{A}$ is the remote server that hosts a database uploaded by $\mathcal{C}$. All items in the database are indexed and can be accessed concurrently by the clients.

1. $\mathcal{A}$ and $\mathcal{C}$ engage in polynomial rounds of the following query-response based protocol
   (a) $\mathcal{A}$ selects two sets of item indices $\mathcal{O}_1 = \{x_1, x_2, \dots, x_n\}$ and $\mathcal{O}_2 = \{y_1, y_2, \dots, y_n\}$ such that $x_i \neq x_j$ and $y_i \neq y_j \forall i, j \in [1, n]$. $\mathcal{A}$ sends $\mathcal{O}_1(i)$ and $\mathcal{O}_2(i)$ to c_i where $\mathcal{O}_j(i)$ is the i^{th} item in $\mathcal{O}_j$.
   (b) On the basis of a fairly selected bit b, the clients query for the items in $\mathcal{O}_b$.
   (c) Observing the queries in Step 2, $\mathcal{A}$ outputs bit $b^{'}$
   (d) $\mathcal{A}$ wins the round iff. $b^{'} = b$.
2. $\mathcal{A}$ wins the security game iff. she can win any round with non-negligible advantage over random guessing where non-negligibility is defined over an implementation specific security parameter.

A mechansim that satisfies the above security game ensures that it protects against the two aforementioned leaks. To see why consider the following:

Correlation between data items accessed in a single round. The security game straightforwardly ensures that $\mathcal{A}$ can win a round with non-negligible advantage over guessing if she can distinguish between two sequence of concurrent accesses. Therefore, a mechanism that satifies the game ensures that an adversary cannot correlate items concurrently accessed in the same round.

Correlation between data items accessed in successive rounds. Consider two randomly chosen sets of item $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{O}_1$. In two successive rounds, $\mathcal{A}$ provides the following sets of items to $\mathcal{C}$:

- Round1: $\mathcal{O}_1$ and $\mathcal{O}_2$
- Round2: $\mathcal{O}_1$ and $\mathcal{O}_3$

In this case, if $\mathcal{C}$ queries for $\mathcal{O}_1$ both in round 1 and 2, and $\mathcal{A}$ could correlate accesses in the current round with previous rounds, $\mathcal{A}$ can win round 2 with non-negligible advantage. Therefore, a mechanism that satisfies the security game also ensures that an adversary cannot correlate accesses in successive rounds.

ConcurORAM gaurantees multi-client concurrent access security in context of the security game by ensuring access pattern privacy of concurrent accesses in each round using PD-ORAM and RingORAM as described in Section 4.

4. OVERVIEW

Consider a database with N blocks of data shared among c clients. ConcurORAM stores the data blocks in a RingORAM on the server and the position map entries for the RingORAM in a server hosted pyramid ORAM (PD-ORAM). The stash for the RingORAM (which holds blocks that overflow from the RingORAM tree) is also stored on the server. Since, the stash can grow (and shrink) dynamically after an eviction, this allows the server to learn the number of blocks that have been evicted to the tree during a RingORAM eviction. To prevent this leak, ConcurORAM allocates a fixed sized stash which is equal to the maximum size that the stash can grow as determined by the experimental upper bound in [15].

Position map. Each position map entry is indexed by a logical block ID and contains the corresponding leaf ID in the RingORAM tree to which that data block is mapped, or indicates that the block is in the stash. Since each level of PD-ORAM contains multiple fixed-sized buckets, an entry for a particular logical block ID, if it exists in a PD-ORAM level, is located in the bucket determined by applying the secret hash function for that level on the logical block ID. Clients query for a position map item by downloading the corresponding buckets from each level of PD-ORAM as determined by the hash function for that level. Further, the top level of PD-ORAM is designed to contain as many entries as the size of the stash. This ensures that during an eviction, even if the entire stash is evicted to the tree, the entries for these blocks can be added to the top level of PD-ORAM.

In addition, the server maintains three append-only logs: query log, position map (PM) result log and data result log. The logs are stored encrypted on the server.

Query log. The query log records all currently ongoing transactions. Clients append the logical ID of the data block they are querying for and download the query log to check for overlapping accesses. In the case where the required data block is already being accessed by another client (and there is a previous entry in the query log for the same), the client proceeds with a dummy access (described later). The query log ensures that all clients have a consistent view of ongoing transactions and thus do not query for the same block.

PM result log. The PM result log contains items that have been accessed from the position map in the last round of concurrent accesses. After reading an entry from the position map, a client reencrypts and return the entry to the server which is appended to the PM result log.

Data result log. The data result log contains the data blocks that have been accessed from the RingORAM in the last round of concurrent accesses. After finding the required block from either the requested path in the RingORAM tree or the stash, a client reencrypts and return the item to the server which is appended to the data result log.

Figure 1 shows the query protocol for ConcurORAM . Queries proceed as follows:

1. In step 1a and step 1b, a client appends the logical

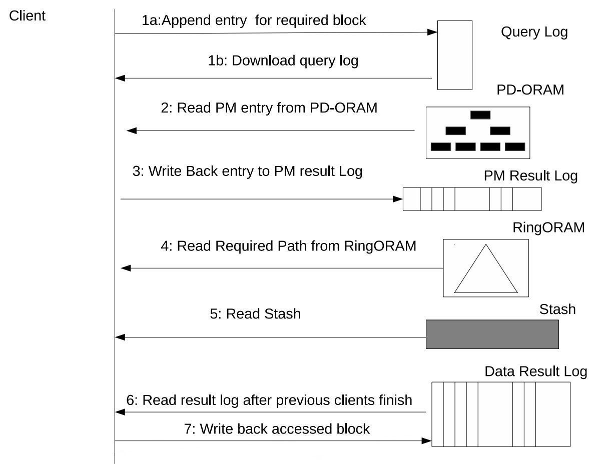
Client
1a:Append entry for required block
Query Log
1b: Download query log
PD-ORAM
2: Read PM entry from PD-ORAM
3: Write Back entry to PM result Log
PM Result Log
RingORAM
4: Read Required Path from RingORAM
Stash
5: Read Stash
Data Result Log
6: Read result log after previous clients finish
7: Write back accessed block

Figure 1: Overview of a query. Steps 1-6 can proceed concurrently. In step 7, clients wait for previous clients to finish before downloading data result log

block ID of the required block and downloads the query log. If the required block is already being accessed, the client proceeds with a dummy access in the following steps.

2. In step 2, the client reads a bucket from each level of PD-ORAM to locate the position map entry for the required logical block ID. The PD-ORAM access protocol ensures that the client finds the required position map entry in this step. For a dummy access, the client reads a random bucket from each level of PD-ORAM.

3. In step 3, the client reencrypts and writes back the actual position map entry read in Step 2 to the PM result log. For a dummy access, the client writes a "fake" item to the PM result log (such as an encryption of all zeroes).

4. In step 4 and 5, the client reads a block from each bucket on the RingORAM path on which the queried block exists (as determined from the position map entry) and the stash. For a dummy access, the client reads dummy blocks from a randomly selected path.

5. In step 6, the client reads the data result log after all previous clients finish. If a client has executed a dummy query, the client gets the required block from the data result log.

6. If a client found the required block in step 6, the client writes back the updated value of the block to the data result log for a write access. Otherwise, the client reencrypts and writes back the accessed block.

Note that in step 7, a client finds the most updated version of the required block after all previous ongoing transactions accessing the block has completed.

Eviction. After a fixed number of accesses, the logs are cleared through an eviction to the RingORAM tree (Figure 2). More specifically, after k (a fixed parameter) concurrent queries, the client with the last query reads the logs, the stash and a predetermined path from the RingORAM tree (chosen in the reverse lexicographical order) and tries to evict blocks in the data result log and the stash to the path. First, the data result log and the stash are merged. Since, there may be multiple compies of the same block in the data result log due to concurrent accesses for the same block, only the latest copy of a block in the data result log is merged with the stash. The client tries to evict as many blocks from the stash and data result log combined to the path. The overflow from the eviction forms the new stash.

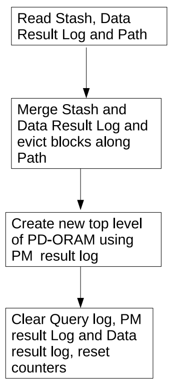
Read Stash, Data Result Log and Path
Merge Stash and Data Result Log and evict blocks along Path
Create new top level of PD-ORAM using PM result log
Clear Query log, PM result Log and Data result log, reset counters

Figure 2: High level description of the eviction process. The path to be evicted is chosen in the reverse lexicographical order.

Then, the client creates the new top level of PD-ORAM with the new mappings for the evicted blocks. Since, the top level is of fixed size and the number of blocks evicted is variable, the client adds "fake" entries to ensure that the new top level is of the same size irrespective of the outcome of the eviction. This is similar to Privatefs [21], where the result log becomes the top level of PD-ORAM after a fixed number of accesses. Further, all the logs are cleared after the eviction.

The server maintains an access counter and ensures that the eviction takes place after a fixed number of concurrent accesses. During the eviction, the server locks all client accesses to ensure consistency. Note that only in step 7 of the query protocol, clients need to wait for previous clients to finish. Moreover, since client queries proceed concurrently, the queries are expected to finish at almost similar times.

5. REFERENCES

[1] *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008* (2008), The Internet Society.

[2] Bindschaedler, V., Naveed, M., Pan, X., Wang, X., and Huang, Y. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM,

pp. 837–849.

[3] Chung, K., Liu, Z., and Pass, R. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. *CoRR abs/1307.3699* (2013).

[4] Devadas, S., Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. *Theory of Cryptography: 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, ch. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM, pp. 145–174.

[5] Gentry, C., Goldman, K. A., Halevi, S., Julta, C., Raykova, M., and Wichs, D. *Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, ch. Optimizing ORAM and Using It Efficiently for Secure Computation, pp. 1–18.

[6] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *Journal of the ACM 43* (1996), 431–473.

[7] Goodrich, M. T. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM 58*, 6 (Dec. 2011), 27:1–27:26.

[8] Goodrich, M. T., and Mitzenmacher, M. Mapreduce parallel cuckoo hashing and oblivious RAM simulations. *CoRR abs/1007.1259* (2010).

[9] Goodrich, M. T., and Mitzenmacher, M. Privacy-preserving access of outsourced data via oblivious ram simulation. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II* (Berlin, Heidelberg, 2011), ICALP'11, Springer-Verlag, pp. 576–587.

[10] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2011), CCSW '11, ACM, pp. 95–100.

[11] Islam, M. S., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *in Network and Distributed System Security Symposium (NDSS* (2012).

[12] Kushilevitz, E., Lu, S., and Ostrovsky, R. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms* (2012), SIAM, pp. 143–156.

[13] Moataz, T., Mayberry, T., and Blass, E.-O. Constant communication oram with small blocksize. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 862–873.

[14] Pinkas, B., and Reinman, T. Oblivious ram revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology* (Berlin, Heidelberg, 2010), CRYPTO'10, Springer-Verlag, pp. 502–519.

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

[16] Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT* (2011).

[17] Stefanov, E., and Shi, E. Oblivistore: High performance oblivious cloud storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 253–267.

[18] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 299–310.

[19] Wang, X., Chan, H., and Shi, E. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 850–861.

[20] Williams, P., Sion, R., and Carbunar, B. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 139–148.

[21] Williams, P., Sion, R., and Tomescu, A. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 977–988.