ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM

Anrin Chakraborti*, Radu Sion*
*Stony Brook University, {anchakrabort, sion}@cs.stonybrook.edu

***Abstract*—ConcurORAM is a parallel, multi-client oblivious RAM (ORAM) that eliminates waiting for concurrent stateless clients and allows overall throughput to scale gracefully, without requiring trusted third party components (proxies) or direct inter-client coordination. A key insight behind ConcurORAM is the fact that, during multi-client data access, only a subset of the concurrently-accessed server-hosted data structures require access privacy guarantees. Everything else can be safely implemented as oblivious data structures that are later synced securely and efficiently during an ORAM "eviction". Further, since a major contributor to latency is the eviction – in which client-resident data is reshuffled and reinserted back encrypted into the main server database – ConcurORAM also enables multiple concurrent clients to evict asynchronously, in parallel (without compromising consistency), and in the background without having to block ongoing queries.**

As a result, throughput scales well with increasing number of concurrent clients and is not significantly impacted by evictions. For example, about 65 queries per second can be executed in parallel by 30 concurrent clients, a 2x speedup over the state-of-the-art [13]. The query access time for individual clients increases by only 2x when compared to a single-client deployment.

I. Introduction

As increasing amounts of confidential data are outsourced in today's cloud-centric environments, providing confidentiality and privacy becomes critical. To ensure confidentiality, outsourced data and associated metadata can be encrypted client-side. Data remains encrypted throughout its lifetime on the server and is decrypted by the client upon retrieval. However, encryption is simply not enough for ensuring confidentiality since *access patterns* may leak significant information [8].

Oblivious RAM (ORAM) allows a client to hide data access patterns from an untrusted server hosting the data. Informally, the ORAM adversarial model ensures indistinguishability between multiple equal-length client query sequences. Since the original ORAM construction by Goldreich and Ostrovsky [6], a large volume of literature [12, 14, 15, 16] has been dedicated to developing more efficient ORAM constructions. Of these, under an assumption of $\mathbb{O}(n)$ client storage with small constants, PathORAM [14] is widely accepted as asymptotically the most *bandwidth efficient* ORAM. RingORAM [12] further optimizes PathORAM for practical deployment by reducing the bandwidth complexity constants.

Although tree-based ORAM designs [12, 14] have achieved near-optimal *bandwidth* for single-client scenarios, one critical challenge remains un-addressed, namely the ability to accommodate multiple concurrent clients efficiently. It is straightforward to deploy existing schemes to support multiple clients by sharing ORAM credentials and storing data structures that would normally be maintained client-side (e.g., the stash and the position map in the case of a tree-based ORAM) on the server to ensure state consistency across multiple clients. However, in such a setup, to maintain access privacy, only *one client* can be allowed to access the server-hosted data structures at any one time. This reduces the overall throughput and significantly increases the query response time. A client may need to wait for *all* other clients to finish before retrieving a data item. Since ORAMs often have non-trivial query latencies, this usually results in significant access latency for a client before being able to proceed with the query.

An existing line of work on parallel ORAM constructions [1, 3, 5, 7, 11] achieve parallelism at no additional bandwidth cost under the assumption of constant inter-client awareness and communication. Although a step forward, this poses barriers that are often times difficult to handle in real scenarios. Without inter-client communication, Taostore [13] assumes a trusted close-to-server (proxy) to achieve parallelism. All client requests are routed to the proxy, which deploys multiple threads to fetch one (or more) paths from a PathORAM [14] data tree and satisfy numerous client requests at once. The need for a trusted third-party however deviates from the standard ORAM model, where trust is only placed on the clients at most. Moreover, a trusted proxy may be difficult to deploy in reality as well as presenting a single point of failure/compromise.

Motivation. This paper addresses these shortcomings by eliminating the need for trusted proxies and inter-client communication, and allowing client queries to proceed *independently* in the presence of other ongoing queries. *Thus, ConcurORAM is the first to achieve parallelism for stateless ORAM clients in the standard trust model without the need for direct inter-client communication.*

A. Challenges and Key Insights

Asynchronous Accesses. Tree-based ORAMs feature two different classes of accesses to the server: (i) *queries* (reading

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23411
www.ndss-symposium.org

a root-to-leaf path) and *evictions* (writing back some of the previously read data items to the root-to-leaf path). A common strategy for reducing overall bandwidth/round-trips is to couple queries and evictions [14] . Even if a multi-client design can be envisioned in which metadata is stored on the server for consistency, this coupling forces a synchronous design in which only one client is in charge at any given time.

One approach for decoupling accesses is to maintain and update a locally-cached subtree and smartly sync with on-server data structures without blocking queries [13]. Since, the local subtree essentially performs the role of a write-back cache, the construction does not immediately scale to a multi-client setting. This can be resolved by having a designated trusted client (a close-to-server proxy) maintain the subtree locally and route queries to the server. However, the assumption of a trusted proxy introduces several performance and security drawbacks. The most important of these is that the system's overall performance now entirely depends on the proxy's resources e.g., available network bandwidth. An under-provisioned proxy or a system failure/compromise will adversely affect all clients in the system. Also, this design does not support stateless clients (or storage-limited clients) – the proxy carefully synchronizes accesses based on (potentially large amount of) metadata stored locally. Outsourcing this metadata naively may lead to privacy leaks.

To eliminate this security/performance bottleneck and allow practical interactions with asynchronous decoupled multi-client operations, ConcurORAM adopts a different approach. Queries only perform non-blocking read-only accesses. Further, evictions write back changes with access privacy in the background to additional server-hosted oblivious data structures (ODS) (append-only logs, write-only tree etc.). These data structures are designed to be synchronized with the main ORAM tree periodically and efficiently. The synchronization mainly involves copying contents between server-hosted data structures, reference swaps etc. which can be performed securely and efficiently with limited client interaction, optimizing both bandwidth and round-trips.

Parallel Queries. Asynchronous accesses alone cannot facilitate parallel query execution. The query protocols for tree-based ORAMs are fairly complex and involve multiple read/write-back steps. Without careful synchronization, overlapping accesses will violate consistency and privacy.

Possible solutions are parallel query abstractions such as in PrivateFS [16], which protects inter-client query privacy for any multi-client "non-simultaneous" ORAM. As we will see, in ConcurORAM, this forms the basis of a parallel sub-query mechanism, suitably modified to support parallel evictions.

Parallel Evictions. While parallel query execution is a good starting point, evictions are at least as expensive as the queries. Thus, even with parallel queries, serialized evictions will be the de facto bottleneck and limit overall throughput gains.

ConcurORAM overcomes this limitation by allowing multiple evictions to execute in parallel. In essence, this is possible because we observe that when evictions are performed according to a *deterministic eviction schedule* [12], and the number of parallel evictions is fixed, access patterns to the server-hosted data structures can be clearly and deterministically defined.

This allows identifying the critical sections of the eviction protocol where synchronization is necessary, and the design of associated fine-grained locks. The remainder of the protocol can be performed in parallel using additional server-hosted data structures designed to maintain state and enforce minimally-sufficient global synchronization.

Evaluation. As we will see, because the critical sections are small, this results in an overall throughput that scales gracefully with increasing number of concurrent clients and is not significantly impacted by evictions. For example, about 65 queries per second can be executed in parallel by 30 concurrent clients with only a 2x increase in query access time over a single-client deployment. Importantly, this is a 2x speedup over the state-of-the art [13], which operates under stronger assumptions of a trusted proxy.

II. Related Work

ORAMs have been well-researched since the seminal work by Goldreich and Ostrovsky [6]. We specifically discuss existing parallel ORAM constructions here and refer to the vast amount of existing literature for further details on general ORAM construction [6, 12, 14, 15, 16].

Oblivious Parallel RAM (OPRAM). Boyle *et al.* [1] first introduced an oblivious parallel RAM (OPRAM) construction assuming inter-client communication for synchronization. Clients coordinate with each other through an *oblivious aggregation* operation and prevent simultaneous clients from querying for the same block. For colliding client accesses, only *one representative* client queries for the required item while all other clients query for dummy items. The *representative* client then communicates the read item to all other colliding clients through an *oblivious multi-cast* operation. Subsequent works [3, 4, 5, 7, 11] have optimized Parallel RAMs matching the overhead of a sequential ORAM construction.

TaoStore [13]. Another interesting parallel ORAM construction is TaoStore which achieves parallelism for PathORAM. through a trusted proxy. All client queries are redirected to the trusted proxy which then queries for the corresponding paths from the PathORAM data tree. Further, the proxy runs a secure scheduler to ensure that the multiple path reads do not overlap and leak correlations in the underlying queries. TaoStore achieves a significant increase in throughput but can support only a limited number of parallel clients before the throughput plateaus due to the proxy's bandwidth constraints.

PD-ORAM [16]. Williams et al. provided a parallel ORAM construction that does not require trusted proxies and inter-client communication. However, the construction is derived from a hierarchical ORAM construction, with higher access complexity than standard tree-based ORAMs. Hence, the overall throughput gain is limited.

III. Background

A. Oblivious RAM

Oblivious RAM (ORAM) is a cryptographic primitive that allows a client/CPU to hide its data access patterns from an untrusted server/RAM hosting the accessed data. Informally, the ORAM adversarial model prevents an adversary from distinguishing between equal length sequences of queries made by the client to the server. This usually also includes indistinguishability between reads and writes. We refer to prior works for more formal definitions [12, 14, 15, 16].

B. PathORAM

PathORAM is an efficient ORAM construction with an overall query asymptotic access complexity of $\mathbb{O}(\log N)$ blocks, matching the known lower bound [6]. PathORAM organizes data as a binary tree. Each node of the tree is a *bucket* with multiple (constant number of) blocks. A block is randomly mapped to a unique path in the tree.

> *Invariant: A block mapped to a path resides either in any one of the buckets on the path from the root to the corresponding leaf, or in a* stash *that is stored locally*

Position Map. PathORAM use a "position map" data structure to map logical data item addresses to identifiers of tree leafs defining a corresponding path from the root, "within" which the data items are placed. 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. The position map is either stored on the client ($\mathbb{O}(N)$ client storage) or recursively in smaller ORAMs on the server.

Access. To access a particular block, the client downloads all the contents along the root-to-leaf path 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 [12, 14]. We specifically describe the RingORAM [12] protocol as a building block.

C. RingORAM

RingORAM [12] is derivative of PathORAM [14] that optimizes practical bandwidth requirements. This is the result of two optimizations: i) de-coupling the queries from the expensive eviction procedure, and ii) fetching only *one* block from each bucket in the tree during queries.

Unlike PathORAM [14], where a query needs to fetch all buckets along a path from the root to a particular leaf, RingORAM query cost is independent of the bucket size. This is achieved by storing additional dummy blocks in each bucket. Bucket-specific metadata tracks the locations of blocks (and dummy blocks) within buckets. Each query first reads this metadata and determines whether the required block is present in a particular bucket. A dummy block is fetched from the buckets that do not contain the required data block.

The additional dummy blocks makes the RingORAM buckets larger than PathORAM buckets. This makes evictions expensive. To overcome this, RingORAM delays evictions by de-coupling queries and evictions – an eviction is performed after a fixed number of queries. To make evictions more effective, Ring ORAM uses a deterministic eviction schedule based on the reverse-lexicographical ordering of leaf IDs to select eviction paths.

Query. The query protocol in RingORAM is as follows

1) Determine the path to which the block is mapped using the position map.
2) For each bucket on the path
   - Use the bucket-specific metadata to determine if the required block is in that bucket.
   - If the block is in the bucket, read the block. Otherwise, read a dummy block.
3) Download the entire stash (if stored on the server). Add the queried block to the stash (if not already present).
4) Remap the block to a new randomly selected path, update position map accordingly.

Evictions. After a fixed number of queries, an eviction is performed as follows

1) Download the contents of an entire path from the tree determined by the *reverse-lexicographical ordering of the leaf IDs*. Place the contents in the stash.
2) Write back as many blocks as possible from the stash (re-encrypted) to a new locally created path.
3) Write back the contents of the new path to the tree

Deterministic Selection of Eviction Paths. RingORAM selects eviction paths by ordering the leaf identifiers in the *reverse-lexicographical order*. Specifically, the reverse lexicographical representation denotes each path of the data tree as a *unique* binary string. The least significant bit (LSB) of the string assigned for a path is 0 if the target leaf corresponding to the path is in the left subtree of the root and 1 otherwise. The process is continued recursively for the next bits, up to the leaf, with the next bit(s) being assigned based on whether the leaf is in the left or right subtree of the children nodes.

Intuitively this results in better evictions by spreading out blocks uniformly across the tree since consecutive eviction paths have minimum overlaps, and N paths of the tree are selected once before the same path is selected again.

Access Complexity. ORAMs are typically evaluated in terms of *bandwidth* – the number of *data blocks* that are downloaded/uploaded in order to complete one logical request. RingORAM features an overall bandwidth of $\mathbb{O}(\log N)$ data blocks, where N is the total number of blocks in the ORAM. This asymptotic bound holds only under the *large block size assumption* when the data blocks size is $\Omega(\log^2 N)$ bits.

ConcurORAM has the same large block size assumption and all access complexities reported in this paper indicate the number of physical blocks that are accessed overall for a fulfilling a particular logical request.

IV. Overview

A. Preliminaries

Trust Model. There are two types of parties: the ORAM clients (with limited local storage) and the ORAM server (a remote storage hosting client data).

- *Honest-but-curious server*: The server can observe all requests and attempts to correlate them by saving and comparing snapshots. The server does not deviate from the ConcurORAM protocol.
- *Trusted clients*: Clients are *honest* and share secrets (credentials, keys, hashes etc.) required for accessing the ORAM. *Clients do not need to interact with each other, but can observe and track other client accesses through the server-hosted data structures.*

Server Storage. As with most tree-based ORAMs, the main server-side data structure is a binary tree storing fixed-sized data blocks. Specifically, a database with N logical blocks requires a binary tree with N leaves.

Node Structure. ConcurORAM follows the same node structure as Ring ORAM [12]. Specifically, each node of the tree contains a fixed number of data blocks (denoted by Z) and dummy blocks (denoted by S), collectively referred to as a bucket. Note that $(Z + S) \in \mathbb{O}(\log N)$ blocks. Blocks in a bucket are encrypted with semantic security (e.g., using randomized encryption) and randomly shuffled. Buckets also store relevant metadata for retrieving specific data blocks, identifying dummy blocks etc.

Stateless Clients. In addition to the main data tree, tree-based ORAMs require several auxiliary data structures for maintaining state. This includes a *position map* to track the locations of logical blocks in the ORAM data tree and a *stash* to hold blocks that could not be immediately placed back on the tree (due to randomized placement). Typically, these are stored client-side to speed-up accesses. However, in a multi-client setting, this metadata needs to be stored persistently on the server, in order to present a consistent view for all clients.

Temporary Client Storage. Clients require a small amount of temporary storage to perform several operations locally before uploading updates to the server. This storage is bounded by $\mathbb{O}(\log N)$ block for a database with N blocks.

Building Blocks. For illustration purposes, we will consider the RingORAM query and eviction protocols (described in Section III-C) as a (non-parallel) starting point for some of the ConcurORAM protocols.

We note however that this is not necessary – the techniques presented here can be generalized for other tree-based ORAMs provided the following two conditions are satisfied

- The ORAM supports *dummy queries* (where only dummy blocks are fetched from the tree), and the dummy queries are indistinguishable from real queries.
- Eviction paths are selected using the reverse-lexicographical ordering of leaf identifiers.

Position Map Design. For a concurrent ORAM design, the position map itself needs to allow concurrent access to position map entries. In ConcurORAM, this can be realized by storing the position map recursively in smaller ConcurORAM instances. However, this introduces several implementation challenges due to concurrent recursive data structure accesses.

To avoid recursion, a simpler alternative is to store the entire position map in a parallel hierarchical ORAM e.g. PD-ORAM [16], which does not require its own position map.

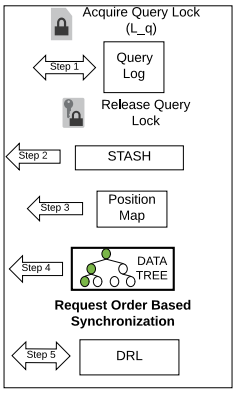


Fig. 1: Parallel query overview. Accesses to the stash, the position map and the data tree can proceed concurrently. Queries access the query log after acquiring a mutex (the query lock) for a short period of time. Synchronizing accesses to the DRL ensures that queries are completed in the order in which they start execution, a necessary condition to prevent security leaks.

In the current design and implementation of ConcurORAM, the position map is stored in a PD-ORAM instance shared by the clients. Since PD-ORAM supports parallel multi-client accesses, the position map is treated as a secure black box.

We specifically note that this approach does not affect overall complexity. Position map entries for N blocks are typically $\mathbb{O}(\log N)$ bits in size. The overall PD-ORAM access complexity is $\mathbb{O}(\log^2 N)$ items. Thus, accessing a position map entry from PD-ORAM has an access complexity of $\mathbb{O}(\log^3 N)$ bits, or $\mathbb{O}(\log N)$ blocks of size $\Omega(\log^2 N)$ bits. This is equivalent to a data access for tree-based ORAMs.

Server Functionalities. Without the aid of explicit inter-client communication or trusted proxies, achieving a notion of global synchronization in ConcurORAM requires server-managed mechanisms and APIs for fine-grained locking, all while guaranteeing access privacy and obliviousness.

This is different from prior work [13] where inter-client synchronization is handled by the proxy and the server is treated basically as a storage device, only providing APIs for downloading, uploading, copying and deleting data.

B. Parallel Queries

Supporting parallel queries is the first step towards achieving full parallelism. On observation, it may be evident that if two clients concurrently query for the same block, according to the query protocol in Section III-C, they will access the same path from the tree, thereby leaking inter-client query privacy. Prior work solves this by either assuming direct inter-client communication and synchronization [1, 5] or by routing queries through a common proxy [13] which executes only the non-overlapping queries.

Without requiring these assumptions, ConcurORAM provides inter-client awareness through server-hosted data structures. The techniques presented here are similar to the parallel query abstraction described in [16] – the goal is to convert an ORAM that is secure for non-parallel queries into an ORAM with parallel queries (Figure 1), augmented with judiciously designed server-hosted data structures.

Query Log. To support concurrent queries without leaking inter-client privacy, information about all ongoing transactions is written to an encrypted *query log*, not unlike a transaction log. Prior to executing a query, clients first download the entire log (to check for overlapping accesses) and then append to the log the encrypted logical address of the data block they are querying for. This ensures that all clients have a consistent view of ongoing transactions. In case of an overlapping query – when there is a previous entry for the same block in the query log – the client proceeds with a *dummy query* by simply reading dummy blocks from a random path in the tree.

Data Result Log (DRL). For overlapping queries, the above ensures that only *one* client can access the target block at any time. The other clients must wait for an eviction to re-randomize the location of the block in the tree before they will access the block. However, this may result in indefinite wait times and possibly leak privacy under a timing channel [13].

To mitigate, ConcurORAM caches previously accessed blocks in a *data result log* until *periodic* ORAM evictions can place them back to random locations on the data tree.

At the end of an access, the block queried by a client is re-encrypted and appended to the DRL. Other clients that queried for the same block (but ended up performing a dummy query instead) can then access the block by reading the entire DRL.

Request Order-based Synchronization. Since clients do not communicate, it is not possible for a particular client to learn when its target block is in the DRL. Further, if clients accessed the DRL *only* in case of dummy queries, the server will be able to distinguish overlapping queries for the same block. The following DRL access protocol resolves all these concerns

1) Clients executing a query read the DRL after all "previous" clients finish their queries. Specifically, queries that started execution earlier by registering an entry in the query log, must complete *all* steps of the query protocol before the current client can read the DRL. The client checks this by comparing the size of the current DRL and the ordering of entries in the query log. If the client registered the i^{th} entry to the query log, it can proceed only after there are $i-1$ blocks in the DRL. Note that if a client executed a dummy query, then it will necessarily find the target block in the DRL after all previous queries have finished execution.
2) After reading the entire DRL, a client always appends the (updated and re-encrytped) target block to the DRL.

Query Round & Log Size. To bound the log sizes, ConcurORAM introduces the notion of a *query round*. Specifically, only up to c (a constant) queries are allowed to execute in parallel, while requests that arrive later have to wait until all executing queries finish – this includes appending their results to the data result log. A round of c parallel queries constitutes a query round in ConcurORAM. Queries arrive and execute individually after registering an entry in the query log, but belong to the same query round as long as the current query log contains less than c entries. Once all queries in the current query round finish execution, the contents of the DRL (which now contains c blocks) are evicted back to the data tree and the DRL and the query log are cleared. This ensures that the query log and DRL size never exceed c entries.

C. Non-blocking Evictions

Parallelizing the query step alone is not sufficient to achieve good scalability. Evictions are often expensive and can block clients for possibly impractical amounts of time. Instead, to scale, ConcurORAM performs evictions continuously in the background ensuring that queries are blocked for very short upper-bound periods of time. Achieving this is not straightforward. We need to introduce several key insights.

First, note that in the non-parallel case, queries and evictions (Section III-C) include the following client-side steps

- *Eviction*
  1) Fetch buckets from a path of the data tree.
  2) Evict contents of the stash to new path locally.
  3) Write back the new path and stash.
  4) Update the position map and clear the logs.
- *Query*
  1) Update the query log.
  2) Read the stash and position map query.
  3) Fetch the query path from the data tree.
  4) Read and update the data result log.

Observe that eviction Steps 1 and 2 perform only read accesses and do not conflict with the query protocol. Eviction Step 3 however updates the data tree and the stash and requires synchronization to avoid inconsistencies.

Insight 1: Separate Trees for Queries and Evictions. One way to to synchronize this is to *perform queries on a read-only copy of these data structures while the data tree and stash updates during evictions happen on a writable copy never accessed by the actual queries*. Once eviction completes updating the writable copies, their contents can be (efficiently and securely) copied (by the server) to the read-only data tree and stash version, and made available for future queries. This is aided by two server-hosted data structures:

- *Write-only tree*: A write-only tree ("W/O tree") is initialized with the same contents as the read-only data tree. The W/O tree is updated during evictions while the data tree is used to satisfy queries in the background. This is possible because queries do not update the data tree and queried blocks that are updated with new data make it back to the data tree only during evictions.
- *Temporary stash:* During the eviction of blocks along a specific path of the write-only tree (eviction path), blocks that cannot be accommodated are placed in a "temporary stash" (not accessible to queries) on the server.
  Once the entire eviction path is updated, the eviction path is copied from the write-only tree to the data tree, and the temporary stash is made available for queries by efficiently replacing (e.g., by a simple reference swap) the main stash with the temporary stash. Note that at this stage the contents of the main stash have already been evicted to the data tree and the temporary stash, and thus can be replaced without losing track of data.

Insight 2: Multi-phase Evictions. To execute evictions without blocking queries, ConcurORAM splits evictions into

- *Processing Phase*
  1) Fetch eviction path buckets from the *write-only* tree
  2) Fetch current stash
  3) Evict contents of the stash and the eviction path
  4) Create a new path and temporary stash locally
  5) Write back the updated path to the *write-only* tree
  6) Write back the temporary stash to the server
- *Commit Phase*
  1) Update the position map for the blocks that have been evicted to the *write-only* tree
  2) Copy the eviction path from the *write-only* tree to the data tree (server-side copy)
  3) Swap reference of the main stash (previously used to satisfy queries) with the temporary stash
  4) Clear the query log
  5) Clear the data result log

An eviction can perform the processing phase in its entirety before performing the commit. Eviction processing – which is significantly more expensive than the commit – can be executed in parallel with queries. This allows ConcurORAM to block queries only while an eviction *commits*.

Insight 3: An Oblivious Data Structure for Storing and Privately Accessing Query Results. Since queries do not need to wait for evictions, multiple query rounds may run by the time one eviction completes. Blocks accessed in these query rounds need to be stored somewhere until an eviction subsequently replaces them on the tree. Each round of parallel queries that is executed in the background while an eviction takes place, generates a *data result log* (DRL) containing the blocks that have been accessed by its queries.

These DRLs need to be maintained separately on the server until their contents can be evicted back into the data tree (which can then be accessed by future queries). Further, reading all such DRLs in their entirety in the query protocol (Step 5 in Figure 1) may be too expensive. Instead we propose a mechanism to efficiently query for particular items from the DRLs without leaking their identity to the server.

To this end, DRLs that are pending evictions are stored in an oblivious data structure, namely the *DR-LogSet*. This allows *clients to efficiently query for particular blocks from the DRLs without leaking to the server: (i) the identity of the block, and (ii) the block's last access time.*

D. Parallel Evictions

One challenge here is that if blocks are evicted to the data tree from a single DRL at a time, the DR-LogSet may grow uncontrollably – by the time a DRL is cleared, several new DRLs will have been created.

To ensure that the DR-LogSet remains bounded to an acceptable size, evictions must start as soon as a DRL has been added, even if a previous eviction has not finished. Effectively, evictions must execute in parallel. One obvious roadblock here is that multiple evictions cannot commit simultaneously

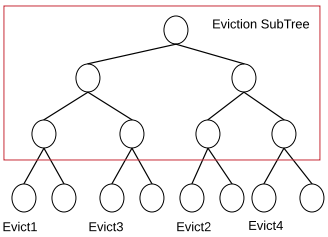


Fig. 2: Eviction subtree (EST) defined by 4 parallel eviction clients evicting to paths in reverse lexicographical order of their leaf IDs.

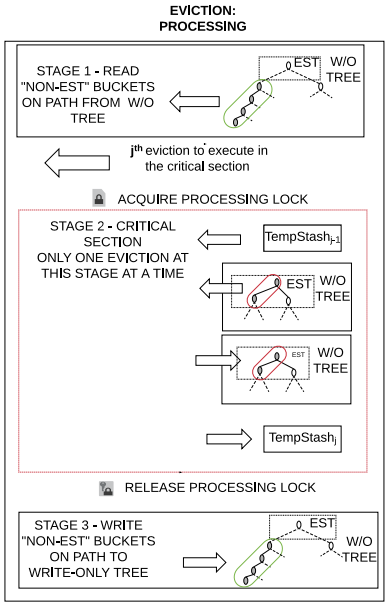


Fig. 3: Eviction processing is divided into three stages. Fixing the maximum number of evictions that can be executed in parallel at initialization allows ConcurORAM to define a maximal sub-tree outside of which all ongoing evictions cannot overlap, namely the EST. Stages 1 and 3 read and write back the non-EST buckets from eviction path, respectively. An eviction executing Stage 1 does not overlap with evictions in Stage 3. Stage 2 is the critical section which updates the EST and is accessed after acquiring the processing lock. The execution of the critical section defines an ordering of evictions, which is used later to serialize commits.

because during the commit phase, the same data structures need to be updated. As we will see next, a viable, efficient approach is to perform the (expensive) processing phases in parallel while serializing the commits.

Insight 4: Identifying Critical Sections for Parallel Eviction Processing. Facilitating parallel eviction processing is challenging due to overlapping accesses to the *write-only* tree in Step 5 of the processing phase: two independent paths being evicted to in parallel will invariably intersect at some level of the write-only tree. Updates to any buckets residing on the paths' intersection need to be synchronized. A key insight here is that we can precisely predict the overlaps! This is because evictions are performed to deterministically selected paths.

First, recall from Sections IV-A and IV-D that evictions are performed to data tree paths in reverse lexicographical order of their corresponding leaf IDs. Since each path is represented by a *unique* lexicographical representation, *any k consecutive eviction paths are deterministic*. Further, if

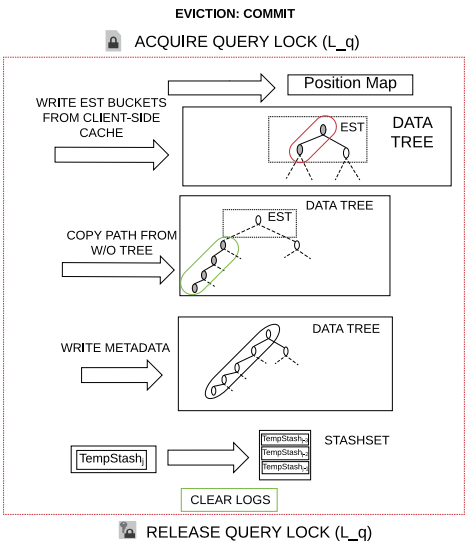


Fig. 4: Eviction commit includes updating the position map, updating the data tree path and adding the temporary stash to the StashSet. Only one eviction can commit at a time, while also blocking queries during commit.

only these k (determined based on system load) evictions are allowed to execute in parallel, then the overlaps between the paths can be predicted precisely. As a result, we can *define a maximal subtree outside of which* any k *consecutive eviction paths will never overlap*.

- The *eviction subtree* (EST) is a subtree of the write-only tree containing the root and the buckets overlapping between any k paths corresponding to consecutive evictions allowed to execute in parallel. For k consecutive evictions executing in parallel, the height of the EST is $h = \log k + 1$ (Section V-B). Figure 2 shows an example EST with 4 parallel evictions.
- *Fine-grained locking for eviction subtree access.* Accesses to the eviction subtree constitute the critical section of the *processing phase* and is protected by a mutex, namely the *processing lock*. These accesses must be performed *atomically* by a single eviction client at a time. Multiple evictions can execute the rest of the steps of the processing phase in parallel.

Insight 5: Asynchronous Commits. Updates performed in the critical section of the processing phase determine the behavior of future evictions and queries. An eviction that enters the critical section uses results generated by previous evictions. Thus, *if commits are serialized based on the order in which evictions enter the critical section, additional synchronization is not necessary* to ensure consistency of data structures. However such fully serialized commits may result in unnecessary and prolonged waits.

To further increase parallelism, ConcurORAM allow asynchronous eviction commits. Concurrent evictions can commit in any order. To enable this, information from each eviction is securely persisted server-side and reconciled later when all preceding clients in the serial order also finish their commits. This is facilitated using an oblivious data structure:

- The *StashSet* is similar in design and functionality to the DR-LogSet. In particular, the StashSet stores the temporary stashes for out-of-order commits. Critically, the StashSet also allows oblivious queries – *clients can efficiently query for particular blocks in the StashSet without leaking to the server: (i) the identity of the block, and (ii) the last access time of the block.*.

V. Technical Description

Notation. The server stores N blocks of data, with logical address in $[0, N-1]$. A *real data* block with logical address id is denoted by b_{id}. Dummy blocks are assigned addresses (just for reference) outside the address space of real data blocks. A dummy block with address i is denoted as d_i. Once a data block is retrieved from the server, it is uploaded back only after re-encryption with fresh randomness.

Temporary Identifiers. Prior to execution, both queries and evictions are assigned *temporary identifiers*, which are extensively used for synchronization without inter-client communication (as will be discussed later). Specifically

- *Query identifier:* Each query in a round is logically identified by a unique *query identifier* $0 \leq i \leq c-1$. The query identifier reflects the order in which the queries start execution by appending an entry to the query log. Clients learn their query identifiers prior to query execution from the query log.
- *Eviction identifier.* ConcurORAM synchronizes evictions based on the critical section. Specifically, the order in which evictions execute the critical section, is also used for ordering the commits, for the sake of consistency. For this, a *processing counter*, is stored server-side to track the number of evictions that have executed the critical section since initialization, . While in the critical section, an eviction reads the value of this processing counter, which becomes its *temporary eviction identifier*. Before exiting the critical section, the processing counter is incremented.

A. DR-LogSet

The DR-LogSet is used to store and privately query DRLs generated by previous query rounds until an eviction can write back contents to the data tree. The DR-LogSet includes the current DRL (of size c), and $k \leq c$ DRLs generated by rounds of queries that finished execution while an eviction was being performed in the background.

- *Bigentry logs.* Except for the current DRL, all other previously generated DRLs are stored in the DR-LogSet as "bigentry logs" of size $2 \cdot c$ – each bigentry log is composed of a random permutation of the (re-encrypted) blocks of a previously generated DRL combined with an additional c random dummy blocks. Dummy blocks are assigned identifiers from 0 to $c-1$.
- *Temporal ordering & log identifiers.* Bigentry logs are ordered by ascending insertion time and assigned identifiers, $l_0, l_1, \ldots, l_{k-1}$, with l_{k-1} being the ID of the log inserted most recently.

Algorithm 1 readLogSet(id)

```
Read current DRL
i ← Current DRL size
for j ∈ [k, k − 1, . . . , 1] do
    index_blk ← Read search index for l_j
    if id ∈ index_blk then
        if id ∉ DRL then
            Read b_id from l_j
            Remove id from index_blk
        else
            Read dummy block d_i from l_j
            Remove b_id from index_blk
    else
        Read the d_i dummy block from l_j
    Reencrypt and write back index_blk
```

Algorithm 2 writeLogSet(blk, i)

```
blk ← Queried block to be appended to DRL
Append encrypted blk to current DRL
Read l_i from DR-LogSet
l_i.reshuffle(rand)
Write back l_i to temporary workspace
if i = c − 1 then
    if less than c bigentry logs in DR-LogSet then
                        (Create new bigentry log)
        j ← Number of logs in the DR-LogSet
        Initialize l_{j+1} with size 2c blocks
        l_{j+1} = blk + drl_curr + dummy
        l_{j+1}.reshuffle(rand)
        Append search index to l_{j+1}
        Reencrypt and write back l_{j+1} to the DR-LogSet
        Initialize empty DRL for new round of queries
    else
                    (Wait until next eviction commit)
```

- *Search index:* A search index is appended to each bigentry log to allow retrieval of specific blocks efficiently. This is simply an encrypted list of block IDs ordered by their corresponding positions in the bigentry logs. Due to small-sized entries, the list is small and can be downloaded entirely per access to determine the location of real/dummy blocks in a particular bigentry log.

Querying the DR-LogSet. The readLogSet(id) (Algorithm 1) protocol takes as input the logical ID id of the block being queried and performs the following steps

1) Read the current DRL.
2) *Privately query bigentry logs:* Read *one* block each from the bigentry logs in *descending order* of log ID – recent logs before old ones.
   a) If the queried block ID id is present in a bigentry log as determined from the corresponding search index. and the block is not present in the current DRL, it is retrieved from the bigentry log and ID id is removed from its search index..
   b) If the block is also present in the current DRL, a dummy block d_i is read from the bigentry log and id is removed from the search index.
   c) If the queried block is not in the bigentry log, a dummy block d_i is read.

Periodic Shuffling. The intuition behind the query protocol is to ensure that the server does not learn the identity of the block being queried. Since each of the bigentry logs contains c dummy blocks, they need to be shuffled once every c accesses thus ensuring that it does not run out of *unique* dummy blocks. This is performed efficiently in the background as part of an update which reshuffles and writes-back a bigentry log to a temporary *write-only workspace* on the server. *The reshuffled bigentry logs in the workspace replace the old versions in the DR-LogSet after a round of c queries (as part of the query protocol, Section V-G).*

Updating the DR-LogSet. The writeLogSet(blk) (Algorithm 2) protocol takes as input: i) the block blk that needs to be appended to the current DRL, and ii) the client query identifier i, and performs the following steps

1) Append blk to the current DRL
2) *Reshuffle specific bigentry log:* Each bigentry log is uniquely identified by its ID $l_i, 0 \leq i < c$. Each client registering a new query in the query log is also uniquely identified in the current query round by its query identifier. The client with query identifier i reshuffles the bigentry log identified by ID l_i
3) *Create new bigentry log:* If this update is performed as the last query in a round of c queries, the client initializes a new "bigentry log" with $2 \cdot c$ blocks. The block queried by the client and the up-to-date blocks from the current DRL are added to the newly created bigentry log while filling up the remaining part with dummy blocks.

Data blocks of the DR-LogSet bigentry logs are eventually evicted to the data tree by future evictions. Specifically, once the current eviction completes, the next eviction will evict blocks from the "oldest" bigentry log. Therefore, if the DR-LogSet already contains c bigentry logs, a new log cannot be added until at least one eviction is completed.

Obliviousness. The DR-LogSet query protocol (Algorithm 1) ensures that the server does not learn: i) the identity of the block being queried, and ii) the last access time of the block.

- A block is read from each of the bigentry logs, in a *specific order*, regardless of the bigentry log that actually contains the block. Due to the random permutation of blocks in each bigentry log, the block read from each log appears random to the server.
- Blocks that have been read once from a bigentry log, have their corresponding entries removed from the search index. Indices are updated client-side and encrypted with semantic security, preventing the server from learning which entry has been removed. As a result, the same block is not read from the same bigentry log ever again.
- Due to the round robin reshuffling,, a bigentry log is accessed c times before it is replaced by an independently reshuffled version of the log (unless the bigentry log is cleared by an eviction before). As each log contains c dummy blocks, a different dummy block can be read for each of the c accesses before the next reshuffle.

Theorem 1. *The accesses to the DR-LogSet produce transcripts that are indistinguishable and independent of the block that is being queried.*

Proof available in full version [2].

B. Eviction Subtree

Reverse Lexicographical Ordering of Leaf Identifiers. Recall from Section IV that evictions are performed to paths in the data tree in reverse lexicographical order of their corresponding leaf IDs. Intuitively this results in all N paths of the tree being selected once before the same path is selected again for eviction. ConcurORAM stores a global counter, ctr, tracking the number of evictions that have been executed since initialization, and the *next* k successive evictions are to paths that have reverse lexicographical representations matching the binary representations of $v = (ctr + 1) \bmod N, \ldots (ctr + k) \bmod N$ respectively. Let these paths be $p_1, p_2, \ldots, p_k$.

Since each path is represented by a *unique* lexicographical representation, any k consecutive eviction paths are deterministic. Further, if only these k evictions are allowed to execute in parallel, then the overlaps between the paths can be predicted precisely. As we show next, if the maximum number of consecutive evictions that can be executed in parallel is fixed at initialization, to say k, we can define a maximal subtree outside of which the k successive eviction paths will never overlap in the write-only tree. We first present a related result.

Theorem 2. *The length of the longest common suffix between the binary representations of any $k \leq N/2$ consecutive integers selected from the integer modulo group of N, i.e. $\{0, 1, \ldots, N-1\}$, is bounded by $\log k$.*

Proof available in full version [2].

By Theorem 2, given any two values within the next k consecutive values of v, say v_1 and v_2, the binary representations of v_1 and v_2 cannot have a common suffix of length greater than $\log k$. On further introspection, it can be observed that the length of the longest common suffix between $v_1, v_2, \ldots, v_k$, corresponds to the levels of the tree where the correspondingly chosen k eviction paths, $p_1, p_2, \ldots p_k$ can possibly intersect.

In effect, k parallel evictions to paths determined by the reverse-lexicographical ordering of leaf IDs that started in succession, can overlap with each other on at most the first $\log k + 1$ levels of the tree.

Definition 1. *The eviction subtree (EST) is a full binary tree of height $h = \log k + 1$, where k is the maximum number of consecutive evictions that are allowed to execute in parallel at any given time, such that the root of the write-only tree is also the root of the eviction subtree*

Critical section: Importantly, while writing back to the write-only tree, updating the EST and uploading the temporary stash is the critical section and is performed atomically by only one eviction at a time, enforced by a mutex (processing lock). The rest of the path can be updated asynchronously.

Fixing the Number of Parallel Evictions. ConcurORAM fixes the maximum number of *consecutive* parallel evictions that can be executed at a time, $k \leq N/2$, during initialization. For e.g., if $e_1, e_2, \ldots, e_k$ are the k consecutive evictions that are executing currently, then, eviction e_{k+1} cannot start execution until eviction e_1 completes The value of k will usually be determined by the system load and in general can be set equal to the query log/DRL size (c). Without inter-client communication, one way to achieve this is by maintaining a server-side log of all ongoing evictions. Specifically,

- *Eviction log.* The eviction log stores information about all currently ongoing evictions. Prior to execution, an eviction client reads the eviction log and appends an entry, only if its eviction path does not overlap outside the write-only tree with any of the ongoing evictions. When an eviction commits, the entry from the eviction log is removed. Accesses to the log are synchronized using a mutex, namely the *eviction lock*. It may be evident that the eviction log performs the same role as the query log.

Algorithm 3 Evict.Process

```
1: ctr ← eviction counter
   // Update eviction log
2: Read eviction log
3: if (ctr − k) ∉ eviction log then
4:     EvictionLock.lock
5:     Append ctr to eviction log
6:     EvictionLock.Unlock
7: else
8:     Wait for eviction ctr − k to finish
   // Processing Stage 1
9: path ← WriteOnlyTree.readPath(ctr)
   // Processing Stage 2 (Critical section)
10: ProcessingLock.lock
11: i ← Temporary eviction ID
12: Read TempStash_{i−1}
13: BucketsFromEST ← Read Buckets from EST that intersect with path
14: path.UpdateBuckets(BucketsFromEST)
15: union = TempStash_{i−1} + DRL + path
16: path = union.EvictToPath
17: TempStash_i ← Blocks left in union + dummy
18: Write back TempStash_i
19: for bkt ∈ path.Buckets do
20:     if bkt ∈ EST then
21:         Write bkt to write-only tree
22: ProcessingLock.Unlock
   // Processing Stage 3
23: WriteOnlyTree.write(path, ctr)
```

Algorithm 4 Evict.SyncCommit(i)

```
1: QueryLock.lock
2: Update position map and metadata on data tree path
3: Copy EST buckets from client-side cache to data tree
4: Copy remaining buckets on eviction path from write-only tree to data tree
5: Stash = TempStash_i
6: Clear query log
7: Clear bigentry log from DR-LogSet
8: QueryLock.Unlock
```

C. Parallel Eviction Processing

Processing Protocol. The parallel eviction processing protocol (Algorithm 3) includes the following steps

1) *Stage 1* – Read the non-EST buckets on the eviction path from the write-only tree.
2) *Stage 2* – A critical section which requires acquiring the *processing lock*. It includes the following substeps

a) Read the temporary stash uploaded by the eviction that *last* executed the critical section, denoted $TempStash_{i-1}$, separately maintained on the server.
b) Read buckets in the EST along the eviction path from the write-only tree.
c) Write back updated EST buckets along the eviction path, and the new temporary stash, $TempStash_i$.

3) *Stage 3* – Write back non-EST buckets on the eviction path to the write-only tree.

Processing Cost.

- *Non-blocking stages:* Stage 1 and Stage 3 read and write back $\mathbb{O}(\log N)$ buckets along the eviction path (excluding the small number of EST buckets) on the write-only tree. As discussed in Section IV-A, buckets contain $\mathbb{O}(\log N)$ blocks. Thus, Stage 1 and 3 both feature an asymptotic access complexity of $\mathbb{O}(\log^2 N)$ blocks. Note that these steps can be performed in parallel without blocking queries and other ongoing evictions.
- *Critical section:* Only *one* eviction can execute in stage 2 at a time. This includes reading and writing back buckets along a path from the eviction subtree. The height of the eviction subtree is $\log k + 1$ (Section V-B). The overall asymptotic access complexity of this stage is $\mathbb{O}((\log k + 1)\cdot\log N)$ blocks. This is significantly less expensive than Stage 1 and 2 for realistic deployment scenarios. In fact, stage 2 becomes expensive only when N parallel eviction are allowed to execute in parallel!

D. Synchronous Commits

Before describing the more complex asynchronous commit mechanism, we present a relatively simple design for *synchronous commits – evictions commit in the order in which they execute the critical section*. The challenge here is to persist eviction-specific changes to the eviction subtree. Specifically after an eviction writes to the eviction subtree, subsequent evictions that execute in the critical section can overwrite these contents before the eviction commits. If the contents of the eviction subtree are directly copied to the data tree as part of the commit, this may lead to inconsistencies.

One possible solution is to locally cache the changes to the eviction subtree in a *client-side cache*, and use this to update the contents of the data tree during commits. Specifically, the *client-side cache* includes: i) buckets in the eviction subtree written in Stage 2 of the processing protocol, and (ii) the temporary stash created during eviction.

Synchronous Commit Protocol. The synchronous commit protocol (Algorithm 4) uses the *temporary eviction identifier*, i, of the eviction and performs the following ordered steps

1) Update the position map and eviction path metadata.
2) Copy contents of the eviction subtree buckets from the client-side cache to the data tree.
3) Copy remaining eviction path from the write-only tree to the data tree.
4) Set the temporary stash from the client-side cache as the stash of the data tree.
5) Clear the query log and the bigentry log corresponding to the eviction from DR-LogSet.

Updating Bucket Metadata. Metadata on the data tree path is updated during a commit, while accounting for the current state of the bigentry logs in the DR-LogSet. Specifically, as blocks from a bigentry log are evicted to the write-only tree during eviction processing, some blocks in the log may be accessed by queries executing in the background. Since, there are more recent copies of these blocks in other bigentry logs, the old copies should not be made available for queries. Subsequent queries for these blocks can access the upto-to-date copies from the more recently created bigentry logs.

Observe that these blocks will already be on the eviction path in the write-only tree by the time an eviction commits. As a result, these blocks will also be included when the contents of the eviction path are copied from the write-only tree to the data tree during the commit.

Instead of explicitly removing these blocks by re-scanning the entire eviction path, which will certainly be expensive, ConcurORAM indirectly invalidates these blocks by not updating their corresponding metadata and position map entries on the data tree path. Consequently, old copies are inaccessible to queries (which reads the position map first to locate a particular block) and are removed from the eviction path during later evictions (using the metadata entries on the path).

Finally, recall that once a block has been accessed from a bigentry log, its corresponding entry is removed from the search index. The search index allows ConcurORAM to identify blocks that have been accessed from the bigentry log while the eviction was executing.

E. StashSet

Storing and Privately Querying Temporary Stashes. With synchronous commits (Section V-D), the temporary stash created by an eviction can straightforwardly replace the main stash and satisfy future queries after the commit. However, this is not the case when evictions commit asynchronously.

For example, consider an ordering of evictions established by the execution of the critical section, $e_{i-1} < e_i < e_{i+1}$. Also, let e_{i+1} commit when e_{i-1} has committed but e_i is yet to commit. In this case, contents that are evicted to a path, p_i by e_i from the temporary stash created by e_{i-1} (denoted by $TempStash_{i-1}$), will not be available for queries until p_i is updated in the data tree. Thus, replacing $TempStash_{i-1}$ with $TempStash_{i+1}$ as the main stash will lead to data loss.

StashSet. To overcome this, instead of storing a single stash, ConcurORAM stores a set of *temporary stashes* which were created and uploaded by evictions that have committed asynchronously before previous evictions could be completed. The StashSet is structurally similar to the DR-LogSet and contains $k \leq c$ temporary stashes, organized as follows

- *Temporary stashes:* Each temporary stash contains upto MaxStashSize real blocks and at least c dummy blocks, where MaxStashSize is the stash size determined according to [12]. Blocks are encrypted and randomly shuffled.

- *Search index:* A search index (list of block IDs) tracks the location of real and dummy blocks. The entire (small-sized) search index is downloaded per access.
- *Temporary stash identifier:* The temporary stashes in the StashSet are identified by the *temporary eviction identifiers* of evictions that created the stashes. For example, if the eviction with temporary eviction identifier k created a temporary stash currently in the StashSet, then the temporary stash is identified as $TempStash_k$

Each temporary stash is periodically reshuffled (exactly once every c accesses) to ensure that unique dummy blocks are available for each access. As in case of the DR-LogSet, the reshuffled versions of the temporary stashes are written to a temporary *write-only workspace*. The reshuffled temporary stashes in the workspace replace the old versions in the StashSet after c queries (as part of queries, Section V-G).

Algorithm 5 readStashSet(id, i)

```
1: Read Stash
2: for TempStash_j ∈ StashSet do
3:     index_blk ← Read index for TempStash_j
4:     if id ∈ index_blk and id not already found then
5:         Read b_id from TempStash_j
6:     else
7:         Read d_i from TempStash_j
8: TempStash_i ← Read ith temporary stash
9: TempStash_i.reshuffle(rand)
10: Write-back TempStash_i to temporary workspace
```

Querying the StashSet. readStashSet (Algorithm 5) takes as input: i) the logical address id of the block to query, and ii) the client query identifier i and performs the following

1) Read current Stash.
2) *Privately query temporary stashes:* Read *one* block each from the temporary stashes in *descending order* of temporary stash identifiers
   - Determine if $id \in$ index block of $TempStash_j$
   - If $b_{id} \in TempStash_j$, then read b_{id}
   - Otherwise, read dummy d_i.
3) Reshuffle temporary stash, $TempStash_i$.

Obliviousness. readStashSet ensures that the server does not learn: i) the identity of the block being queried, and ii) the last access time of the block.

- One block is read each from each of the temporary stashes in a specific order, regardless of the temporary stash that actually contains the required block. This prevents the server from learning when the block was added to the StashSet. Due to the random permutation of blocks, the (encrypted) block read from a temporary stash appears random to the server.
- Within a single round of c queries, there can be only one query for a particular block b since if two parallel clients want to access the same block, only one client issues a real query while the other client issues a dummy query.
- Each stash contains at least c dummy blocks, therefore unique dummy blocks can be read for each of the c accesses in a single query round from a temporary stash before a random reshuffle breaks correlations.

Theorem 3. *The accesses to the StashSet produce transcripts that are indistinguishable and independent of the block that is being queried.*

Proof available in full version [2].

F. Asynchronous Eviction Commits

Algorithm 6 Evict.AsyncCommit(i)

```
1: QueryLock.lock
2: Update position map and metadata on data tree path
3: for bkt ∈ EST do
4:     if bk.timeStamp < i then
5:         Copy bkt from client-side cache to data tree
6: Copy remaining buckets on eviction path from write-only tree to data tree
7: Add TempStash_i to StashSet
8: Clear query log
9: Clear bigentry log from DR-LogSet
   // Replace main stash with the temporary stash if this eviction executed
   the critical section earliest
10: if i < j, ∀j ∈ StashSet then
11:     Stash = TempStash_i
12: QueryLock.Unlock
```

The asynchronous commit protocol (Algorithm 6) uses the *temporary eviction identifier*, i, and performs the following

1) Update the position map and metadata on data tree path.
2) Sync data tree:
   - *Copy contents of EST buckets from client-side cache to data tree* – One critical difference here from the synchronous protocol (Algorithm 4) is that instead of copying all the EST buckets from the client-side cache to the data tree, only a part of the path may need to be updated based on the state of commits. This is because a subset of the EST buckets may have been already updated by more recent evictions that accessed the critical section later, but committed earlier. These buckets are in their most recent state and need not be updated during the commit. Buckets can be identified by storing the *temporary eviction identifier* of the eviction that last updated the bucket as part of the bucket metadata.
   - Copy remaining eviction path from write-only tree to data tree.
3) Add temporary stash to the StashSet.
4) Clear the query log and the bigentry log corresponding to the eviction from the DR-LogSet.
5) *If required, set the temporary stash as the main stash:* If this eviction executed the critical section before all other ongoing evictions, then set the temporary stash as the main stash. This ensures *synchronous* updates to the main stash – evictions update the main stash in the order in which they execute the critical section.

Commit Cost.

- *Updating metadata:* Due to the small metadata size ($\mathbb{O}(\log^2 N)$ *bits* [12]), updating metadata along the evic-

Algorithm 7 query(id)

1: $QueryLock.lock$
2: Read query log
3: **if** $id \in$ query log **then**
4: Append $dummy$ entry to query log. // Dummy accesses in next steps
5: **else**
6: Append id to query log.
7: $QueryLock.Unlock$
8: $i \leftarrow QueryLog.length\%c$ // query identifier
9: $stash \leftarrow readStashSet(id, i)$
10: $leafID \leftarrow PM.read(id)$
11: $path \leftarrow DataTree.readBlocksFromPath(leafID)$
// Request order based synchronization
12: $BlocksFromLogSet \leftarrow readLogSet(id)$
13: $blk \in BlocksFromLogSet \cup path \cup stash$ // Required block
14: Update blk for writes
15: $writeLogSet(blk, i)$
16: **if** $i = c - 1$ **then**
17: // Replace all bigentry logs in the DR-LogSet with their reshuffled versions in the temporary workspace
18: // Replace all temporary stashes in the StashSet with their reshuffled versions in the temporary workspace
19: Initialize new query log for future queries.

tion path in the data tree has an overall asymptotic access complexity of $\mathbb{O}(\log N)$ blocks.

- *Upadating position map:* With the position map stored in a PD-ORAM [16], an update has an overall asymptotic access complexity of $\mathbb{O}(\log N)$ blocks.
- *Committing changes to EST:* Committing changes to the eviction subtree from the client-side cache has an overall access complexity $\mathbb{O}((\log k+1)\cdot\log N)$ blocks. A simple optimization here is to store the contents of the client-side cache on a designated *server-side cache*. In that case, committing these changes will only include server-side copies, independent of the network bandwidth.
- *Server-side operations:* The commit also copies data and swaps references for several server-side data structures – this adds negligible bandwidth overhead.

G. Query Protocol

The parallel query protocol (Algorithm 7) includes:

1) Read and append entry to the query log.
2) Query position map.
3) Query StashSet (Algorithm 5).
4) Read path from data tree.
5) *Request order based synchronization:* The client waits for previous clients (that registered their query before in the current query round) to finish their queries.
6) Query and update DR-LogSet. (Algorithm 1, 2)

Theorem 4 (Correctness). *The most up-to-date version of a block is found by queries either on the path indicated by the position map, in the StashSet or in the DR-LogSet.*

Theorem 5 (Query Privacy). *The query protocol (Algorithm 7) produces indistinguishable access transcripts that are independent of the item being accessed.*

Proofs available in full version [2].

Resolving Conflict Between Queries and Commits. Note that both the asynchronous commit protocol (Algorithm V-F) and the query protocol (Algorithm V-G) must first acquire the *query lock* in Step 1. Without explicit synchronization between the clients, this can lead to race conditions and indefinite waits. To resolve this, ConcurORAM enforces two simple policies

- *Commits do not preempt queries.* Once a query starts execution by registering an entry in the query log in Step 2, it completes all the steps before an eviction commit can acquire the *query lock*. Specifically, when an eviction wants to commit, it checks the number of queries that have started execution (size of the query log) and the number of queries that have completed (size of the current data result log). If there are pending queries, the eviction waits for them to finish before acquiring the query lock.
- *Queries wait for commits to bound the DR-LogSet.* As a result of the above condition, eviction commits may have to wait indefinitely. To prevent this, before a query begins execution it checks the number of bigentry logs pending eviction in the DR-LogSet. If the the DR-LogSet is full (already contains c bigentry logs), it waits for a commit.

Query cost. In addition to a position map query ($\mathbb{O}(\log N)$ blocks) and reading a block from each bucket along a path in the data tree ($\mathbb{O}(\log N)$ blocks), queries also read

1) The current stash of size MaxStashSize blocks.
2) The current DRL of size c blocks.
3) *One* block from each DR-LogSet bigentry log.
4) *One* block from each StashSet temporary stash.
5) *One* bigentry log to reshuffle.
6) *One* temporary stash to reshuffle.

MaxStashSize $\in \mathbb{O}(\log N)$ blocks [12] and as both the DR-LogSet and StashSet can contain a maximum of c bigentry logs/temporary stashes, $2 \cdot c$ blocks are read in total for steps 3 and 4. Each bigentry log contains $2 \cdot c$ blocks and each temporary stash has MaxStashSize$+c$ blocks. Thus, Step 5 and 6 have an overall access complexity of $\mathbb{O}(\log N + c)$ blocks. The overall query access complexity is $\mathbb{O}(\log N + c)$ blocks.

VI. Experiments

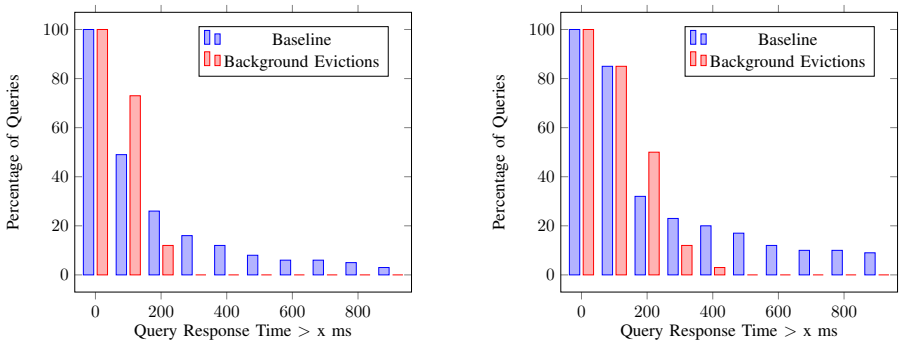
Implementation. ConcurORAM has been implemented in Java. We thank the authors of TaoStore [13] and PrivateFS [16] for providing their implementations for comparison.

Experimental Setup. For all experiments, the server runs on a storage optimized i3.4xlarge Amazon EC2 instance (16 vCPUs and 4x800 GB SSD storage). Clients/proxy run on:

- Bandwidth-constrained scenario: 4 Linux machines on the local network, with Intel Core i7-3520M CPU running at 2.90GHz, 16GB DRAM. The server-client bandwidth is measured as 7MB/s using iperf [9].
- High-bandwidth scenario: Clients run on run on t2.xlarge instance with 4 vCPUs and 16GB of RAM, within the same VPC as the server. In this case, the link bandwidth is measured to be 115MB/s.

A. Asynchronous Accesses for Stateless Client

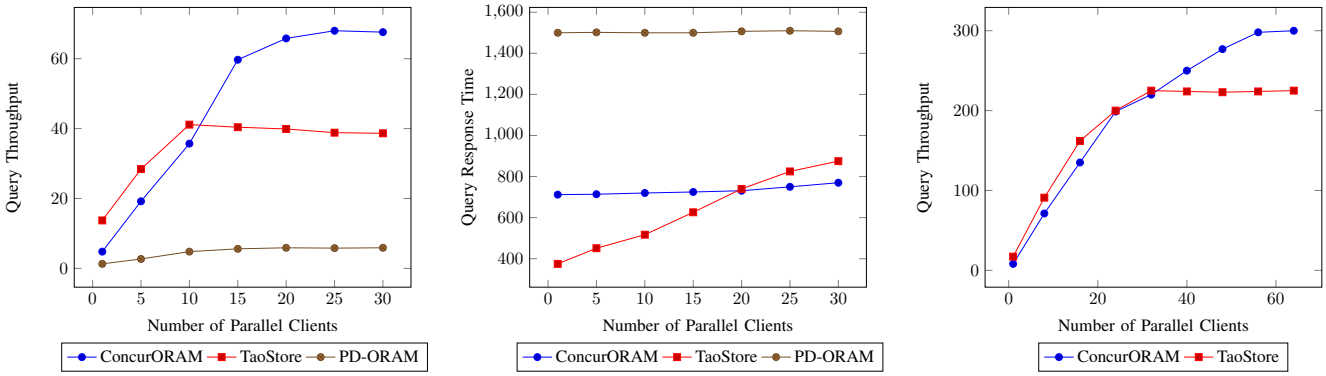
To better understand the benefits of asynchronous accesses in a stateless client setting, we first implemented a version



(a) Query response time distribution. Evictions after 8 queries ($c = 8$)

(b) Query response time distribution. Evictions after 32 queries ($c = 32$)

Fig. 5: (a) The baseline implementation with serial queries and evictions has higher query response times as queries are blocked during evictions. Background evictions bound query response time better. (b) Background evictions remain largely unaffected by the frequency of eviction and help to upper-bound query response times better as evictions become more expensive to support higher eviction frequency



(a) Query Througput (ops/s). Higher is better. B/W = 7MB/s. DB size = 20GB.

(b) Query Response Time (in ms). Lower is better. B/W = 7MB/s. DB size = 20GB.

(c) Query Throughput (ops/s). Higher is better. B/W = 115MB/s. DB size = 20GB.

Fig. 6: (a) TaoStore query throughput plateaus at 10 clients due to proxy bandwidth limitations. ConcurORAM overall throughput scales gracefully up to 30 clients achieving a max. query throughput of 65 ops/s. (b) Query response time for TaoStore increases almost linearly with increasing clients as queries contend for fixed number of proxy threads. Queries from different clients in ConcurORAM and PD-ORAM are independent and thus the query response time remains unaffected by increasing number of clients. (c) With higher bandwidth ConcurORAM can achieve higher overall throughput and plateau only when reaching the server-side limits.

of ConcurORAM with serial queries but equipped with the parallel background eviction support, and compared against a baseline implementation with no background evictions. All metadata (position map, stash etc.) are outsourced to the server.

The parameter of interest is the distribution of query response times. With serial queries, periodic evictions block queries for significant periods of time. In fact, less frequent evictions are not helpful, since this results in a proportional increase in bandwidth due to enlarged buckets.

Figure 5(a) and 5(b) compare the distribution of query response times for ConcurORAM with parallel evictions, against the baseline where evictions are performed serially, blocking the queries. Background evictions bound the query response times better while performance is severely affected by intermittent blocking evictions for the baseline.

B. Parallel Queries with Multiple Clients

We compare parallel query throughput and latency with prior work [13, 16]. Although TaoStore [13] operates in a different trust model, we compare nonetheless to demonstrate the limitations of having a centralized proxy.

If the proxy is over-provisioned, or close to the server (such as within the same Amazon VPC), TaoStore is capable of achieving a high query throughput. But this is seldom the case – an enterprise deploying a (trusted) proxy to route queries will be more likely to place the proxy within its own protected network, rather than deploying it in the same network as the untrusted server, which will at least require trusted execution guarantees etc. Further, placing the proxy near the server will improve proxy performance, but will not necessarily translate to better query throughput for the clients, which will now be constrained by their own link bandwidths with the proxy.

Query Throughput. Our experiments clearly demonstrate this phenomenon. Figure 6(a) shows the overall throughput and the Figure 6(b) shows the average query access times for up to 30 parallel clients for ConcurORAM, TaoStore and PD-ORAM. Clients/proxy are run within our local network, and are thus subject to realistic bandwidth constraints while interfacing with the EC2 server. To conduct experiments with a reasonable number of physical machines, we deploy up to 8 client threads from each of the aforementioned local machines.

Both TaoStore and ConcurORAM outperform PD-ORAM

due to an asymptotically more efficient (by a factor of $\mathbb{O}(\log N)$) base ORAM. Due to the bandwidth limitations of the TaoStore proxy, the throughput for TaoStore plateaus at 10 clients. Expectedly, ConcurORAM can support upto 30 clients without throttling the throughput. At this stage, each of the client machines runs an approximate 8 threads, utilizing the full link bandwidth. Clearly, with independent machines, an even greater number of clients can be supported.

Query Response Time. The query response time for both PD-ORAM and ConcurORAM remain almost constant as clients can query independently. On the other hand, the query response time for TaoStore increases almost linearly with increasing clients as multiple client queries need to contend for the fixed number of query threads deployed by the proxy. At 20 clients, the query response time for TaoStore surpasses the query response time for ConcurORAM. Note that the higher initial query response time for ConcurORAM is because of the stateless design – clients must fetch all metadata including the position map entries and the stash from the server, while TaoStore benefits from storing all metadata on the proxy.

High Bandwidth Scenario. With higher available bandwidth, e.g., when clients and server are within the same network, both ConcurORAM and TaoStore benefit from appreciable overall throughput increase. (Figure 6(c)). Even in this setting, ConcurORAM supports a larger number of clients with a higher overall throughput compared to TaoStore. In fact, as we measure, the plateau observed at around 60 clients, is primarily because the large number of client threads throttle the server-side compute (CPU utilization at 100%) instead of the bandwidth. With a more compute-optimized server, ConcurORAM can support a higher number of parallel clients.

Choice of Parameters. In the experiments above, we set the query round size $c =$ number of parallel clients. Further, we set the number of consecutive eviction that can execute in parallel, $k = c$. This ensures that all bigentry logs in the DR-LogSet can be evicted to the tree in parallel. Since the value of c impacts query access complexity (Section V-G), the overall through plateaus when increasing c (as shown by Figure 6).

In general, c should be determined experimentally based on network conditions to achieve maximum throughput. With a large number of parallel clients, it is possible that maximum throughput is achieved when $c <$ number of parallel clients. In this case, some parallel queries may need to wait for the completion of one (or more) query rounds.

VII. Extensions

A. Fault Tolerance

In the following, we detail how ConcurORAM can proceed gracefully in the event of system crashes, network failures etc.

Crashes During Queries.

1) *Updating the query log (Algorithm 7, Steps 1 - 4):* If the client fails while updating the query log, it will do so while holding the query lock. After a specified timeout, other clients can simply proceed with their queries.
2) *Prior to adding queried block to DRL (Algorithm 7, Steps 5 - 12):* In this case, the client has not yet updated the current data result log with the result of its query. Recall that clients that started execution later wait for this result. After a specified timeout, the *next* client waiting for access to the data result log can repeat the query on behalf of the failed client.
   Observe that this does not leak privacy – if a query fails, the *next* client (public information), always repeats the same query after a specified time, resulting in exactly the same access pattern.
3) *Reshuffling the bigentry log and temporary stashes:* In this case, the failed client will not write back to the temporary workspace. The client that replaces the old versions with the reshuffled version at the end of a query round, can perform the required reshuffles.

Crashes During Evictions. Evictions can fail at various stages, and although specified timeouts will allow the protocol to proceed, it is important to know whether to *roll back* changes and start afresh, or to *roll through* and continue with the protocol for the sake of consistency.

- *Stage 1:* If the eviction fails in stage 1, which only includes read-only access to the write-only tree, a different client can takeover and start afresh.
- *Stage 2 (critical section):* In stage 2, the eviction will fail while holding the *processing lock*. As a result, subsequent evictions waiting for results from the critical section will not proceed. In case of a crash, the eviction can be restarted by another client after rolling back the changes performed in the critical section. Effectively this means disregarding any changes made in the critical section (to the EST buckets and temporary stash) by the failed client and starting afresh.
- *Stage 3:* At this stage, results generated in the critical section by the failed evictions may have been used by subsequent evictions entering the critical section. Thus, the changes cannot be rolled back. Instead, a different client can continue with the eviction process. This requires one critical piece of information – *the pseudo-random mapping* used by the failed eviction for mapping blocks in the bigentry logs to new paths in the data tree. While in the critical section, this information is added to the *eviction log*, and can be used later by a different client for completing the eviction.
- *Commits:* Another client can perform the commit on behalf of the failed client with some additional information
  - *Server-side cache:* Firstly, all contents of the client-side cache which include the temporary stash and the eviction sub-tree buckets needs to be stored on the server. This can be used by a different client for committing updates, in case of a crash.
  - *Eviction metadata:* The client that performs the eviction on behalf of the failed client has to perform several other tasks. This includes updating the position map for blocks evicted, updating metadata on data tree path, and clearing query and data result logs.
    To perform these tasks, the client requires the follow-

ing additional *metadata*, which can be stored in the *eviction log* while in the critical section: i) the eviction identifier, ii) the eviction path identifier, iii) the logical identifiers of the evicted blocks *and* the paths to which they are mapped in the data tree, and iv) the identifiers for the query log and the data result log.

B. Security Against Malicious Server

As a first step, integrity of the server-side data structures can be ensured using existing techniques e.g., *embedding a Merkle tree in the ORAM tree* [14], storing HMACs over the DR-LogSet, StashSet etc. In a single client settings, the root of the Merkle tree and the HMACs can be cached client side and verified for each access. Unfortunately, in a multi-client setting, these variables need to be stored on the server for consistency, leaving open the possibility of replay attacks.

Specifically, as pointed out by previous work [16], the server could simply replay the query log, presenting different views of ongoing transactions to concurrent clients. As a result, clients querying for the same block will have the same resulting access pattern to the server data structures, leaking inter-client privacy. The server could also replay contents of the server-side data structure along with the Merkle tree root hashes, HMACs etc. Without inter-client communication, the best we can do to prevent this is *fork consistency* [10] – if the server selectively replays the states of server-side data structures and presents different (possibly conflicting) views of the system to different clients, then these views cannot be undetectably unified later.

Protecting Against Query Log Replays. Similar to previous work [16], the key idea here is to store a hash tree over all previous queries on the server. Specifically, for each query, a client updates the hash tree with a new leaf record, which includes: i) the clients' own unique identifier, and ii) the logical address of the block queried.

This record (and the updated root hash) is stored client side and as part of its *next* query, the client verifies that the record exists in the hash tree. Since hashes are unforgeable, the server cannot add a new record to the hash tree and forge the root hash value in case of forking attacks. Further, the clients' unique identifiers being included in the record ensures that even if two clients query for the same block, or update the same data structure contents (as a result of a forking attack), the hash records they generate will be different with very high probability, when using a collision resistant hash function.

Protecting Against Data Structure Replays. Similarly, to protect against replay of data structure states, a server-hosted hash tree can record updates to the *data structure integrity variables*. This specifically includes the Merkle tree root hash for the data tree, and HMACs for the DR-LogSet and StashSet.

The client performing updates to the corresponding data structure stores the record (and root hash) locally and verifies that the record exists in the hash tree for subsequent accesses. Using the hash tree, a client can verify the integrity of the data structures. If the server replays states, the client which performed the last update will have a different view of the data structures (root hashes, HMAC values) from the other clients in the system. These views cannot be merged later on by the server without being detected.

VIII. Conclusion

ConcurORAM is a multi-client ORAM that eliminates waiting for concurrent clients and allows overall throughput to scale gracefully with an increase in the number of clients, *without requiring trusted proxies or direct inter-client coordination*. A major insight behind ConcurORAM is the fact that during data access, only a subset of the server-hosted data structures require parallel access with privacy guarantees. Everything else can be implemented as efficient oblivious data structures that are merged later obliviously during an ORAM eviction. ConcurORAM benefits from a novel eviction protocol that enables multiple concurrent clients to evict asynchronously, in parallel, and in the background without blocking queries.

IX. Acknowledgements

This work is supported by the National Science Foundation under awards 1526707, 1526102 and by the Office of Naval Research. We thank our shepherd, Marina Blanton and the anonymous reviewers for their valuable suggestions.

References

[1] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious parallel ram and applications," in *Theory of Cryptography*, E. Kushilevitz and T. Malkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 175–204.

[2] A. Chakraborti and R. Sion, "ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM," *ArXiv e-prints*, Nov. 2018.

[3] T.-H. H. Chan, K.-M. Chung, and E. Shi, "On the depth of oblivious parallel ram," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 567–597.

[4] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Oblivious hashing revisited, and applications to asymptotically efficient oram and opram," in *Advances in Cryptology – ASIACRYPT 2017*. Cham: Springer International Publishing, 2017, pp. 660–690.

[5] B. Chen, H. Lin, and S. Tessaro, "Oblivious parallel ram: Improved efficiency and generic constructions," in *Theory of Cryptography*, E. Kushilevitz and T. Malkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 205–234.

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

[7] T.-H. Hubert Chan and E. Shi, "Circuit opram: Unifying statistically and computationally secure orams and oprams," in *Theory of Cryptography*, Y. Kalai and L. Reyzin, Eds. Cham: Springer International Publishing, 2017, pp. 72–107.

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

[9] E. . L. B. N. Laboratory, *Iperf*, "https://iperf.fr/".

[10] D. Mazières and D. Shasha, "Building secure file systems out of byzantine storage," in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, ser. PODC '02. New York, NY, USA: ACM, 2002, pp. 108–117.

[11] K. Nayak and J. Katz, "An oblivious parallel ram with o(log2 n) parallel runtime blowup," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1141.

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

[13] C. Sahin, V. Zakhary, A. E. Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 198–217.

[14] 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 *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 299–310.

[15] P. Williams, R. Sion, and B. Carbunar, "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*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 139–148.

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