UNIVERSITY OF CALIFORNIA
SANTA CRUZ

TOWARDS MORE EFFICIENT AND PRACTICAL OBLIVIOUS COMPUTATION

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTERS OF SCIENCE

in

COMPUTER SCIENCE

by

Surya Keswani

June 10, 2022

The Thesis of Surya Keswani
is approved:

Professor Ioannis Demertzis, Chair

Professor Alvaro Cardenas

Towards More Efficient and Practical Oblivious Computation

Surya Keswani
Computer Science and Engineering
UC Santa Cruz
Santa Cruz, CA
sukeswan@ucsc.edu

***Abstract*—This paper details some of the most important oblivious RAM (ORAM) constructions in the field of oblivious computing. The field of oblivious computation is in its infancy, with the first paper published in the late 1980s. In 2011, the first efficient oblivious computing construction, TreeORAM, was created. Since the publication of TreeORAM, researchers have innovated new variations of tree-based ORAMs optimized for different workloads. Oblivious computing is ever-growing, and novel research is being conducted in this field at a rapid pace. This paper summarizes the four foundational oblivious computation constructions (TreeORAM, PathORAM, CircuitORAM, and RingORAM) and experiments with improving these constructions using various techniques.**

***Index Terms*—Oblivious Computation, Oblivious RAMS, Tree-based Oblivious RAMS, ORAM**

I. Introduction

People have been concealing messages for thousands of years to protect sensitive information from prying eyes. Basic forms of encryption can be traced back to ancient civilizations such as the Egyptians, who used hieroglyphics to encrypt messages. Current cryptographic protocols are now used to protect sensitive information in transit and data at rest. Standards such as AES 256 CBC and TLS 1.2 RSA help protect the vast amount of financial, medical, and other sensitive data we store online. Searchable symmetric encryption schemes allow clients to search over encrypted data stores efficiently. Though encryption masks data in cipher text, it does not protect against an attacker watching how, when, and what memory is accessed.

Suppose an attacker is looking at an encrypted medical record database. The attacker is only looking at the memory addresses of encrypted data. Between December 25, 2021, and February 25, 2022, approximately 26.5 million Americans, or about 1 in 12 Americans, had a confirmed positive Coronavirus diagnosis [1]. Knowing this public data, an attacker can infer with a high probability that any encrypted medical file that is requested or updated and re-encrypted with fresh randomness is being updated with a Coronavirus diagnosis. An attacker can infer that these updated records are Coronavirus patients. While this fictitious example may seem improbable, it is not.

Recently, researchers accurately inferred $\sim 80\%$ of the search queries over an encrypted email repository by looking at memory access patterns and having a small amount of general background information surrounding the dataset [7]. Searchable encryption schemes are practical but mention a well defined amount of leakage. This leakage has opened the doors to variety of attacks. Researchers have been able to exploit memory access pattern leakage in numerous other ways to recover the query keywords over search-ably encrypted databases [6]. These techniques include having public domain knowledge over a dataset [9] [2], and file injection attacks [12]. Memory access pattern leakage is a critical security vulnerability that needs to, and is, being addressed by current research.

There is also a growing awareness that three companies, Amazon, Google, and Microsoft, supply 58% of all cloud services worldwide [5]. Organizations with sensitive data, such as proprietary intellectual property, stored with a cloud-based provider, should consider the provider an untrusted third party. By implementing oblivious computation constructions and managing their encryption keys, companies can better protect against curious, or worse yet, malicious cloud providers.

II. Trivial ORAMs

There is an urgent need to hide memory access patterns to protect sensitive data. The goal of an oblivious RAM (ORAM) is to hide memory access patterns. The most straightforward solution would be to download an entire encrypted database of size N to a local trusted client, decrypt all the data, fetch the target data, either reading or writing, and re-encrypt the entire database with fresh randomness. While this solution would hide the memory access patterns from an untrusted server, it is highly inefficient. It requires a client to download a potentially large database, iterate over the database, and re-upload the database back to an untrusted server. Furthermore, this trivial solution would not allow multiple clients to access the database simultaneously.

Another simple solution is to have a trusted hardware enclave (stored server side) act as a proxy for a client. The hardware enclave will iterate over an entire database, re-encrypting each block with fresh randomness. If the operation is a read, the enclave will just encrypt the block with fresh randomness. If the operation is a write, the enclave will need to iterate over the entire database, checking if the data already exists. If the data does exist, the enclave will update the data upon a second sequential scan. Otherwise, the enclave will write data to a dummy block during its second sequential

scan. This solution would not require a client to download and re-upload the database (because of the trusted proxy) but still requires a sequential scan for read operations and two sequential scans for write operations. ORAMs this inefficient will never be used, as the most efficient searchable encryption schemes operate in logarithmic time.

III. TreeORAM

TreeORAM [3], published in 2011, is the first ORAM protocol to use the novel bucket ORAM in binary tree constructions. TreeORAM uses constant client-side storage and $\mathcal{O}(N \log N)$ server-side storage. The novel binary tree construction achieves efficient, practical performance and is conceptually much simpler to understand when compared to its predecessors.

For maximum storage of size N, TreeORAM creates a binary tree with $\log N$ levels, and N leaves for a total of $2N - 1$ nodes. Each node in the tree is a bucket containing multiple data blocks, each labeled with a unique block ID (`uid`). Each bucket for TreeORAM has $\log N$ blocks of data. Data blocks can either contain real data or dummy data. The tree is always maintained to be full using a combination of real and dummy blocks. Each leaf node in the tree is assigned a leaf ID (`lid`). The untrusted server stores this tree construction with every data block encrypted by a trusted client.

A trusted client randomly maps each data block to a `lid` to access the tree obliviously. When a block is mapped to a `lid`, it must be stored on the path towards that leaf. For example, looking at Figure 1, If a a block is randomly assigned a `lid` of 4, this block must be stored in either Node 0, 2, 5, or 11. The client keeps track of this block to `lid` mapping in a position map, which is encrypted and stored on the server. TreeORAM uses a `ReadandRemove` function, which builds this protocol's oblivious read and write functions.

`ReadandRemove`

1) Given a `uid` and `lid` paring from the position map, the client fetches all the nodes on the path `lid` from the server.
2) The client decrypts the path of nodes.
3) The client iterates through the decrypted nodes, looking for the target `uid`.
4) When the client finds the target `uid`, it swaps the target data with dummy data.
5) The client re-encrypts the path with fresh randomness and writes the path of nodes back to the server.

With the above-outlined `ReadAndRemove` function, the client can intuitively perform oblivious reads and writes.

Oblivious Reads:

1) The client fetches the `lid` associated with the `uid` for a block from the position map.
2) The client calls `ReadandRemove` for the `uid` - `lid` pair.
3) With the target data now in hand, the client fetches the root node from the server and decrypts the root node.
4) The client iterates over the root node, finding a dummy block. The dummy block is swapped out with the block the client read and removed from the tree.
5) The root node is re-encrypted with fresh randomness and sent back to the server.
6) The client assigns a new random leaf to the block `uid` and updates the position map with the new leaf identifier (`lid`).
7) The client calls an eviction function, which pushes blocks of data further down into the tree.

Notice that because the target block is written back to the root nodes of the tree, it does not matter which `lid` the block is assigned to, as the root node is on every path in the tree. This will also hold for oblivious writes.

Oblivious Writes:

1) The client calls `ReadandRemove` for the `uid` - `lid` pair. This function call will remove any old value for the `uid`. If there is no `uid` to return from `ReadandRemove`, the function will just return a dummy block.
2) The client randomly reassigns the block to-write a random leaf.
3) The client fetches the root node from the server and decrypts the root node.
4) The client iterates over the root node, finding a dummy block. A dummy block is swapped out for the block that should be written to the tree.
5) The root node is re-encrypted with fresh randomness and sent back to the server.
6) The client calls an eviction function, which pushes blocks of data further down into the tree.

Notice that both the oblivious read and write functions operate in similar ways. The server will see the same operation set.

1) The client will fetch a path from the server. `lids` for blocks are uniformly random, so the paths will be randomly chosen. Even if a block is read or written multiple times consecutively, the block is reassigned to a randomly chosen `lid` for every data access.
2) The client will place the path back onto the server. The path has been re-encrypted with fresh randomness.
3) The client will fetch the root node from the server.
4) The client will place the root node back onto the server. The root node has been re-encrypted with fresh randomness.
5) The evict function is called.

The memory access patterns described above are random and follow the same pattern; thus, an untrusted server cannot tell if data is being written to a tree or read from the tree.

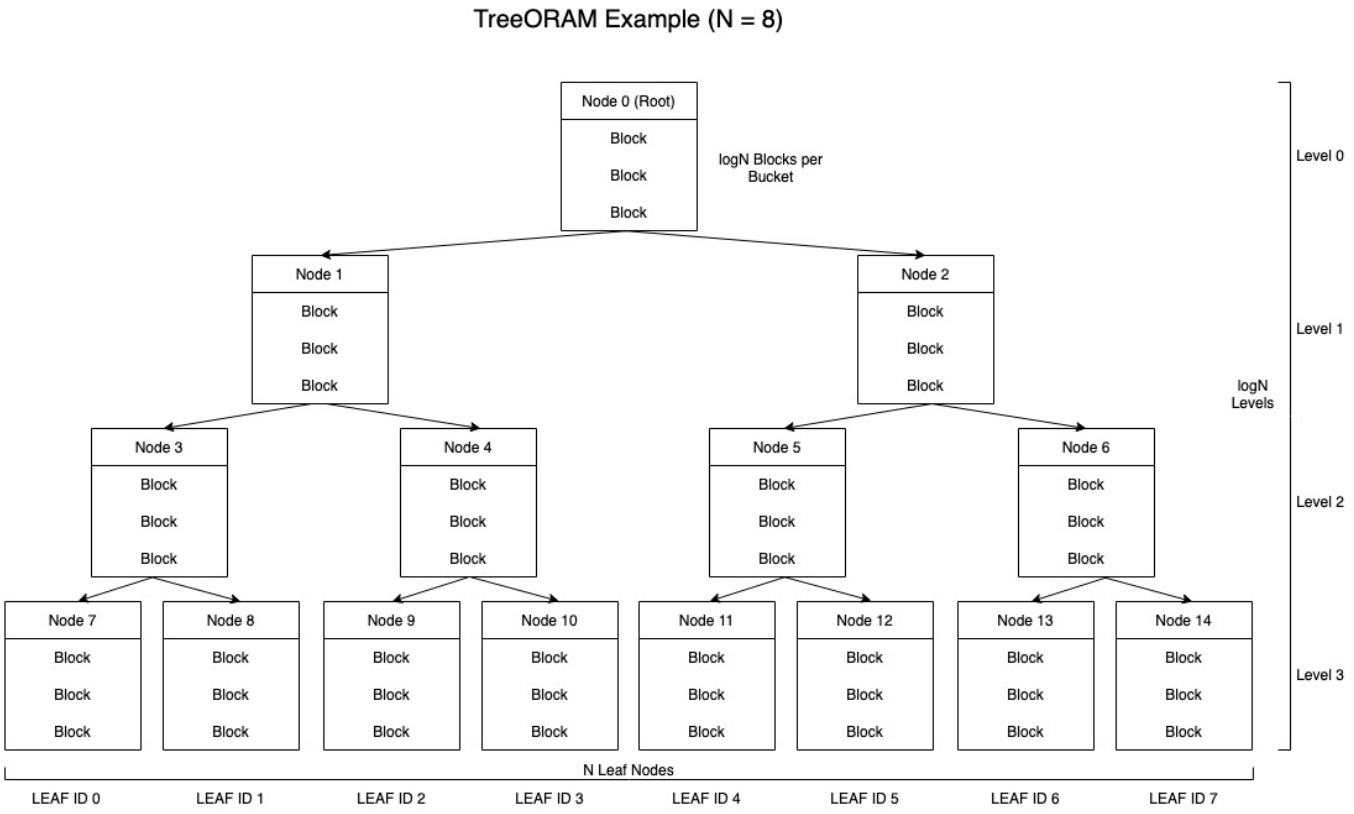

Fig. 1. TreeORAM of size $N = 8$. Blocks can store real or dummy data.

The client writes a data block to the root node for both reads and writes. The bucket at the root node can only store $\log N$ blocks, so continuously writing to the root node will cause an overflow. To ensure nodes in the tree do not overflow, both read and write operations call an eviction function, which pushes blocks of data further down into the tree, freeing up space in the upper levels of the tree.

Two nodes are randomly chosen to be evicted at every tree level (excluding the leaf level). An eviction node will move one data block to each child node. If a node has any amount of real data, one real block and one dummy block will be pushed down to each child node. The real block is pushed down to the child node which is on the path as assigned to the real block by its `lid`. If a node has no real data, two dummy blocks are pushed down, one dummy to each child node. The client fetches eviction nodes from the server and evicts blocks locally to ensure obliviousness. An example of the TreeORAM eviction process can be seen in Figure 2.

TreeORAM runs in $\mathcal{O}(\log^3 N)$. The position map can be recursively stored in a separate ORAM on the server, which runs in $\mathcal{O}(\log N)$. Evictions run in $\mathcal{O}(\log^2 N)$ as $\log N$ buckets per node multiplied by the approximately $\log N$ buckets that are evicted. Evictions match the $\mathcal{O}(\log^2 N)$ run time of reads and writes, which fetch a path size of $\log N$, each with $\log N$ blocks. Combining the cost of position map lookup with an access operation yields a final worst-case run time of $\mathcal{O}(\log^3 N)$ for TreeORAM.

As for storage, the Tree consists of $2N - 1$ nodes, each storing $\log N$ blocks, for a total of $\mathcal{O}(N \log N)$ space. Client-side storage is constant as neither the position map nor the tree is stored locally. Furthermore, the client only needs minimal space when temporarily storing a path locally during computation or running evictions.

IV. PathORAM

PathORAM [4] has a similar bucket ORAM in a binary tree structure as TreeORAM but provides a more straightforward, asymptotically better solution. PathORAM, like TreeORAM, creates a binary tree with $\log N$ levels and N leaf nodes, with each bucket storing Z blocks. Researchers found $Z = 4$ to be an optimal size for each bucket. PathORAM, like TreeORAM, also creates a recursive position map stored on the server. Both constructions also fetch paths to and from the tree based on a leaf identifier (`lid`) associated with each block.

PathORAM, unlike TreeORAM, creates a local storage space known as a stash. The stash is bounded by $\mathcal{O}(\log N)\omega(1)$. PathORAM will fetch an entire path from the server and write to its local stash for data access. This fetched path will leave a path on the server's tree empty. The client will read/write the data locally in the stash. After access, the target block is re-assigned with a randomly generated `lid`. Because PathORAM bounds the stash to a maximum capacity

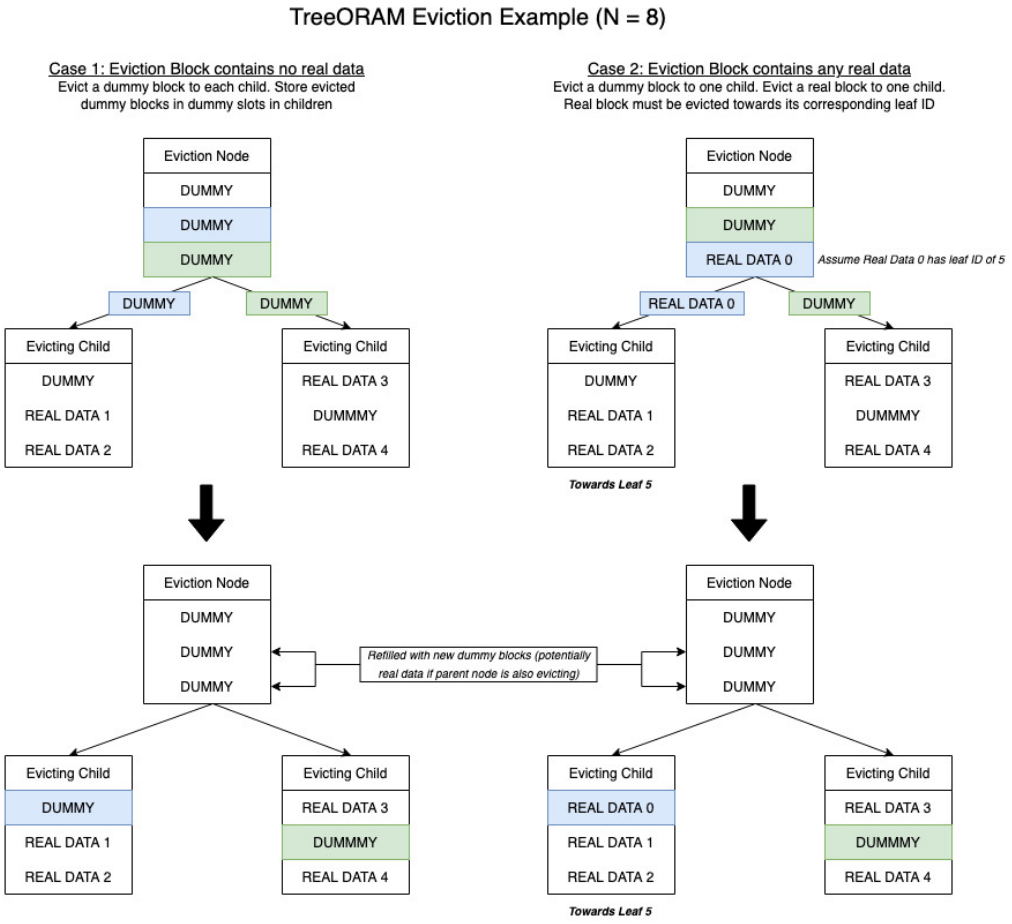

Fig. 2. An example of the eviction process for TreeORAM given $N = 8$.

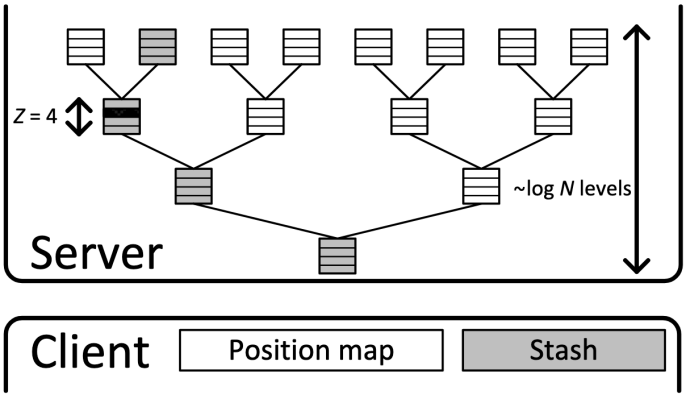

Fig. 3. Assume the shaded black block is mapped to the gray path. The block can be anywhere on the shaded path. 4 Blocks / Bucket. The figure shows a tree of size $N = 8$. This figure is taken from the RingORAM paper. [8]

of $\mathcal{O}(\log N)\omega(1)$, blocks must be evicted from the stash to the tree. When evicting blocks from the stash to the server, PathORAM greedily works from the bottom of the empty path to the top, iterating over the stash and filling the server with blocks from the stash. This new form of greedy eviction provides a stronger guarantee of filling lower levels of the tree first, thus allowing buckets to maintain constant Z storage instead of increasing storage size as the tree size increases. If the stash contains blocks that the client cannot write back to the server due to a lack of space, the stash will store these overflowed blocks until the next eviction cycle. Like TreeORAM, the eviction procedure for PathORAM occurs after ever data access. PathORAM proves that the probability of overflow for the stash is overwhelmingly negligible.

The untrusted server will not be able to tell whether read or write operations are occurring, as it will only see a uniformly randomly chosen path be fetched and put back onto the server. Each path is re-encrypted with fresh randomness, so the server will not be able to tell which data has been placed where based on pattern matching cipher text.

PathORAM runs in a worst-case $\mathcal{O}(\log^3 N)$, with the recursive position map requiring $\log N$ time and evictions cost $\mathcal{O}(\log^2 N)$. The server needs O(4N) space (as Z is 4), and the client stores the stash, which is bounded by $\mathcal{O}(\log N)\omega(1)$. PathORAM has a worst-case bandwidth of $\mathcal{O}(\log N)$, reducing TreeORAM's bandwidth cost of $\mathcal{O}(\log^2 N)$. The small constant Z in PathORAM reduces the time and space complexities

of PathORAM significantly.

V. RingORAM

RingORAM [8] is a variation of PathORAM that is built for workloads that aim to minimize online bandwidth and focus on clients with a small storage capacity. PathORAM transfers path sizes of $Z \log N$ to and from a trusted client. With a standard Z value of 4, PathORAM has an online bandwidth of $4 \log N$ (path from server to client) + $4 \log N$ (path from client to server) = $8 \log N$ (total transportation cost). RingORAM argues that the minimum factor of 8 that PathORAM carries for online bandwidth can increase overhead by $150\times$ for practical parameterizations (cases where the client has gigabytes of storage).

RingORAM, similarly to PathORAM and TreeORAM, creates a binary tree when $\log N$ levels and N leaf nodes. RingORAM stores Z real blocks per bucket and S dummy blocks per bucket. RingORAM maps every data block to a random leaf node, storing this mapping in a position map. RingORAM also creates a stash to store data blocks fetched from the server.

Every bucket of data stores a small amount of additional metadata used to access real blocks quickly. Using the position map, the client will look up a block's leaf node for access. Instead of fetching the entire path from the server, the client will only fetch the metadata for each bucket on the path. The metadata contains information about the real blocks and what the memory addresses for the real blocks are. The client will fetch one block per bucket using the metadata at every bucket. If the metadata does not contain information about the target block, the client will fetch a randomly selected dummy block from the metadata's respective bucket. If the metadata does contain the target block, the client will fetch the target block based on the memory address in the metadata (as seen in Figure 4). Because the metadata is such a small amount of information, it is negligible in the costs of this construction.

As stated, there are S dummy blocks per bucket. If a bucket is accessed S times and does not contain the target data for those S requests, on the $S + 1$ request, the client will reuse a dummy block. This reuse of a dummy block for access leaks information. After every S accesses, a block is written to the client stash to prevent information leakage. The blocks are reshuffled, re-encrypted with fresh randomness, and up to Z blocks from the stash are written back to the server. An additional counter is stored in the metadata of every bucket, keeping track of how many times the bucket has been accessed since its last reshuffle.

RingORAM's improvements over PathORAM require $\log N$ blocks to be fetched instead of $\mathcal{O}(\log^2 N)$ blocks. RingORAM further improves the bandwidth cost using an XOR trick. Instead of sending $\log N$ blocks over the network, the server can XOR all the requested blocks and send one block to the client. Suppose an encryption scheme is used in counter mode. The client can locally compute the cipher text for every dummy block selected and XOR the cipher text away, leaving behind the cipher text for the target block. This XOR trick reduces the online bandwidth to $\mathcal{O}(1)$, thus making this construction significantly more bandwidth efficient than the other tree-based ORAM constructions (as seen in Figure 5).

Like PathORAM, RingORAM must evict buckets from its stash. The updated target block is written to the client's stash for every data read and write. RingORAM follows in PathORAM's footsteps, greedily checking if blocks can be evicted from the stash to the evicting path on the server. The selected path is checked from the leaf level up to the root to ensure evicted blocks are pushed to the lowest levels of the tree. Because buckets in RingORAM can only contain Z real blocks, the eviction process can only fill a bucket with up to Z real blocks. If there are not enough real blocks to evict a bucket, the client will fill the bucket with dummy data.

There are two novel decisions RingORAM makes in its eviction process. Firstly, eviction happens after every A accesses, A being a constant. The choice to perform evictions periodically amortizes the cost of evictions over the A accesses. It is important to note that Z, S, and A are all parameters chosen by the user. The RingORAM paper outlines how to select these three parameters such that the probability of an overflow is overwhelmingly negligible. Secondly, the tree on the server is evicted in a predetermined reverse lexicographic order. A more distributed eviction occurs over the entire structure by evicting the tree in reverse lexicographic order. Refer to Figure 6 for an example of reverse lexicographic order.

VI. CircuitORAM

Oblivious RAM constructions up til a point were primarily focused on reducing the bandwidth, as bandwidth is the primary performance bottleneck for typical client/server models. The creation of tree-based ORAMs points to a promising future of efficient, secure multiparty computation (MPC / SMPC) at scale. The performance focus of SMPC is not bandwidth but rather circuit complexity. The circuit complexity of ORAMs is the total circuit size of the ORAM client over all rounds of interactions. The cost of XOR operations is negligible and is considered essentially free. The primary performance metric of circuit complexity is bound by the number of AND operations.

CircuitORAM [11] follows the same construction as PathORAM with one significant difference, the eviction process. PathORAM's eviction process requires a complex circuit. The intuition behind CircuitORAM is to reduce the circuit complexity of evictions by doing a single scan of the path that is being evicted. CircuitORAM's eviction process picks a single block up at a time and drops the block off at a lower level in the tree. To aggressively evict a single path using a single scan, there must be foresight as to when to pick a block up and drop it down into a lower layer of the path. To gain this foresight, CircuitORAM runs two metadata scans over the eviction path before iterating over the path. The metadata scans provide the aforementioned foresight on when to pick up and drop off blocks. The metadata scans are lightweight and computationally inexpensive, as they only look at a small amount of data, the leaf identifier for each block.

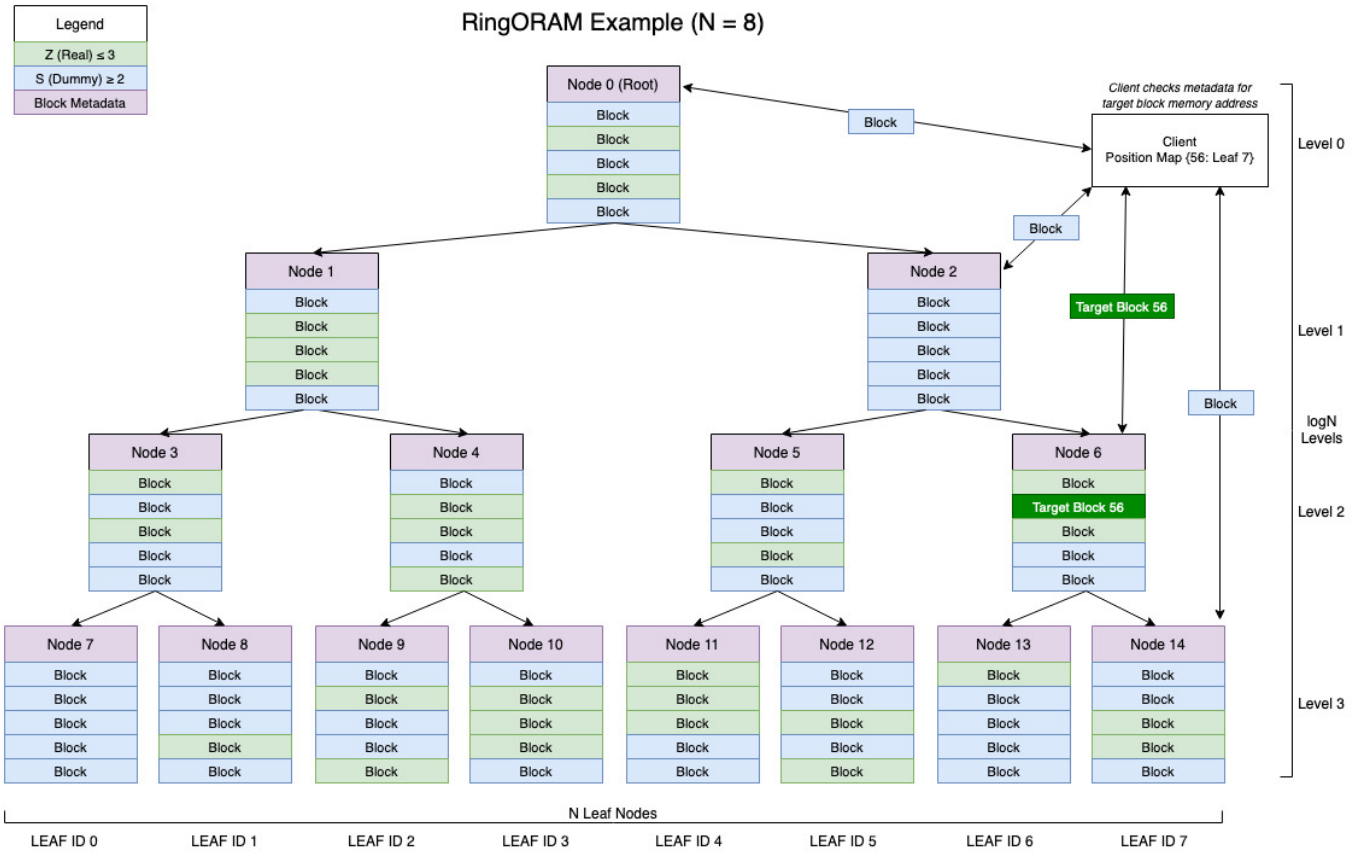

Fig. 4. RingORAM of size $N = 8$. Parameters $Z = 3$, $S = 2$. Example of client fetching blocks without XOR trick.

During the metadata scans, the block data is not touched. The CircuitORAM paper proves a negligible probability of overflow for a bucket size of 2. A detailed description of the metadata scans performed by CircuitORAM is outlined in Figure 8.

VII. Experimental Results

For this master's project, the four discussed ORAMs, TreeORAM, PathORAM, CircuitORAM, and RingORAM were all implemented in C++ [10]. All four implementations follow a general set of similar guidelines. Each implementation is run on a standard laptop, with no client server model. Each implementation is encrypted using AES 256 in CBC mode (using the standard OpenSSL library). A new IV is generated every time a node is encrypted. Nodes all have a data size of 1KB, which is represented by a string of size 1024 bytes. For PathORAM, the author recommended bucket size of 4 is used. For RingORAM, nodes are set to have a Z value of 4, an S value of 3, and A value of 2. Meaning, RingORAM nodes can store a maximum of 4 real blocks, a minimum of 3 dummy blocks, and runs evictions every other (every 2) data accesses. The author recommended bucket size of 2 is used for CircuitORAM.

After completing the four implementations, timing metrics were collected to compare how each of these ORAM constructions perform. Data points were collected for tree sizes: 2^{10}, 2^{12}, 2^{18}, and 2^{20}. Figure 10 shows the total execution time taken by each ORAM. A large cost of the execution time during these experiments was a result of initializing the tree with dummy blocks, as can be seen by the time taken for setup in Figure 11. In these experiments, only 100 data accesses were performed. In practical use cases, an ORAM would be used over a long period of time for many accesses. Thus the setup cost would be amortized over many accesses.

Figures 12 and Figure 13 are the results for access time and eviction time averaged over 100 accesses. Eviction is expensive, overshadowing every other access operation, except for PathORAM. PathORAM's eviction latency is $\sim 33\%$ of its access latency, while for RingORAM, CircuitORAM, and TreeORAM eviction is $\sim 66\%$, $\sim 73.5\%$, and $\sim 80.5\%$ of access latency respectively. TreeORAM is highly ineffective because as the tree size N grows, the size of the bucket grows as well ($\log N$). Setting TreeORAM's bucket size to a constant of 4 improves the eviction latency of TreeORAM by a factor of $\sim 3.9x$. Unfortunately, setting TreeORAM's bucket size to a small constant breaks the security guarantees of TreeORAM and creates a significant probability of bucket overflow, which is unacceptable for a system that is meant to be secure.

Because an ORAM construction would be run on a client and server, and the C++ implementations are all run locally,

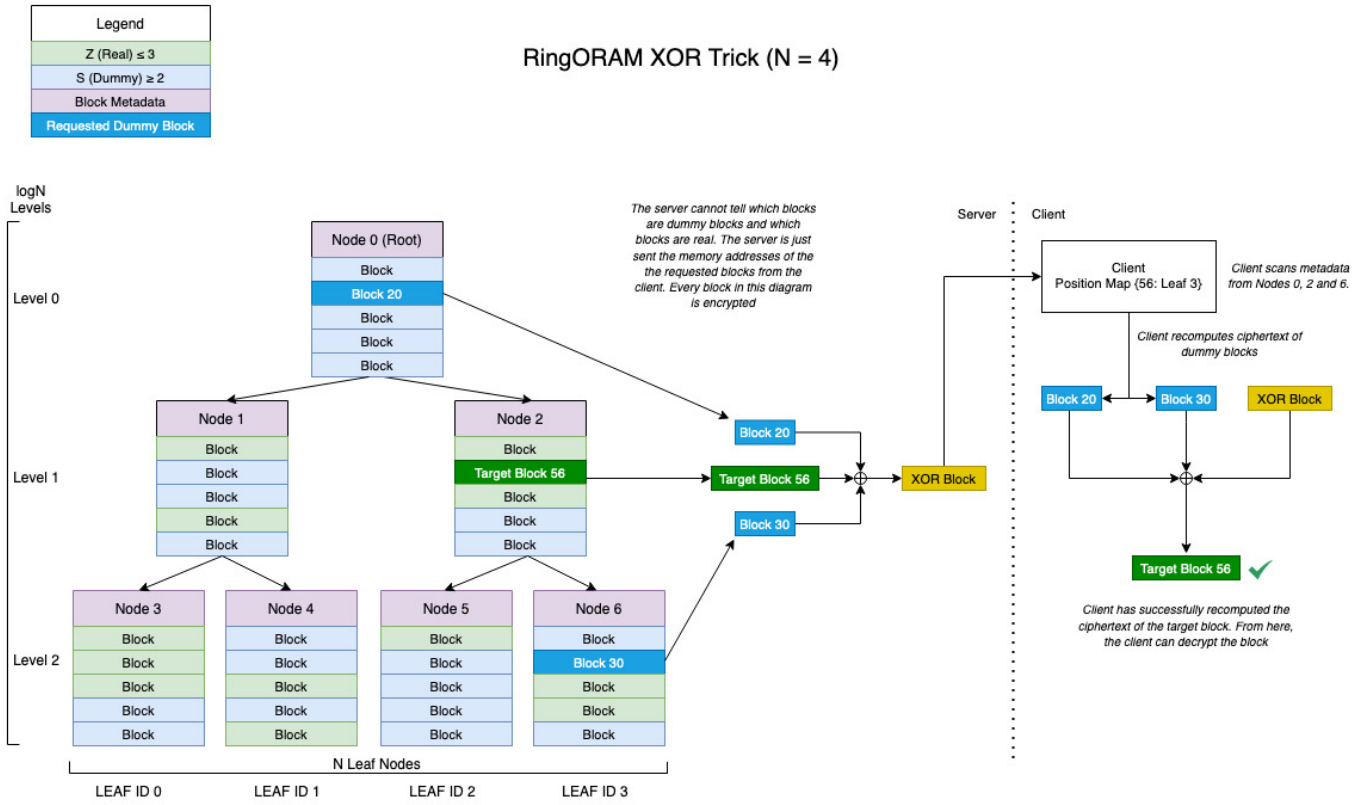

Fig. 5. RingORAM of size $N = 4$. Parameters $Z = 3$, $S = 2$. Example of client fetching blocks with the XOR trick.

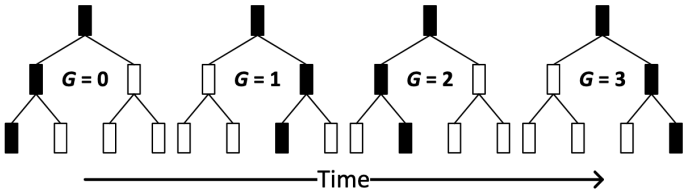

Fig. 6. Reverse lexicographical order of eviction used by RingORAM. This figure is taken from the RingORAM paper. [8]

Tree-based ORAMs		
Binary-tree ORAM [48]	$O((D+\log^2 N)\log^2 N)\omega(1)$	30.1M
CLP13 [8] (naive circuit)	$O((D+\log^2 N)\log^3 N)\omega(1)$	37.9M
CLP13 [8] (w/ oblivious queue [39, 45, 59])	$O((D+\log^2 N)\log^2 N)\omega(1)$	37.9M
Path ORAM (naive circuit) [52]	$O((D+\log^2 N)\log^2 N)\omega(1)$	56.6M
Path ORAM (o-sort circuit) [53]	$O((D+\log^2 N)\log N \log\log N)\omega(1)$	41.4M
Circuit ORAM (This Paper)	$\mathbf{O((D+\log^2 N)\log N)\omega(1)}$	**0.97M**

Fig. 7. Experimental results from the CircuitORAM paper. The columns are as follows: Name of tree-based ORAM construction, asymptotic circuit size, and the number of AND gates used for each construction. Every tree construction has a security failure probability of 2^{-80} and uses 4GB data with a 32-bit block size. The figure is taken from the CircuitORAM paper [11].

a simulated experiment was run to calculate the bandwidth latency of access and evict operations. Figures 14 - 19 display the logarithmic scale of approximate time (in seconds) it would take to perform an access or evict call for the ORAM constructions, at three different speeds (50 Mbps, 300 Mbps, and 1 Gbps). These results were calculated by determining the number of blocks moved back and forth between a client and server. The blocks are set to be 1 KB in size, thus accounting for the size of the data but not the size of any block metadata. Again, these are approximate simulations that do not account for the bandwidth cost of small amounts of block metadata.

VIII. Future Work

After implementing four ORAM constructions, there is a variety of interesting work that can be built on the foundation of this project. One key observation, seen in almost every diagram from the experimental results section, is the inefficiency of the TreeORAM, due in large part to the growing bucket size. It would be interesting to see if an eviction process from another ORAM (Path, Circuit, or Ring) could be applied to TreeORAM to make the construction perform more in line with its peers. Any new eviction process applied to TreeORAM would also require a new security proof, which could allow for a smaller, possibly even constant bucket size. Similarly, it would be interesting to "mix and match" eviction processes among Path, Circuit, and RingORAM and observe how efficiently, or inefficiently, each of these constructions perform.

A similarly interesting idea would be to combine the best of RingORAM and CircuitORAM. These constructions have similar structures and using RingORAM's XOR trick for a constant bandwidth, as well as its hyper parameters for bucket

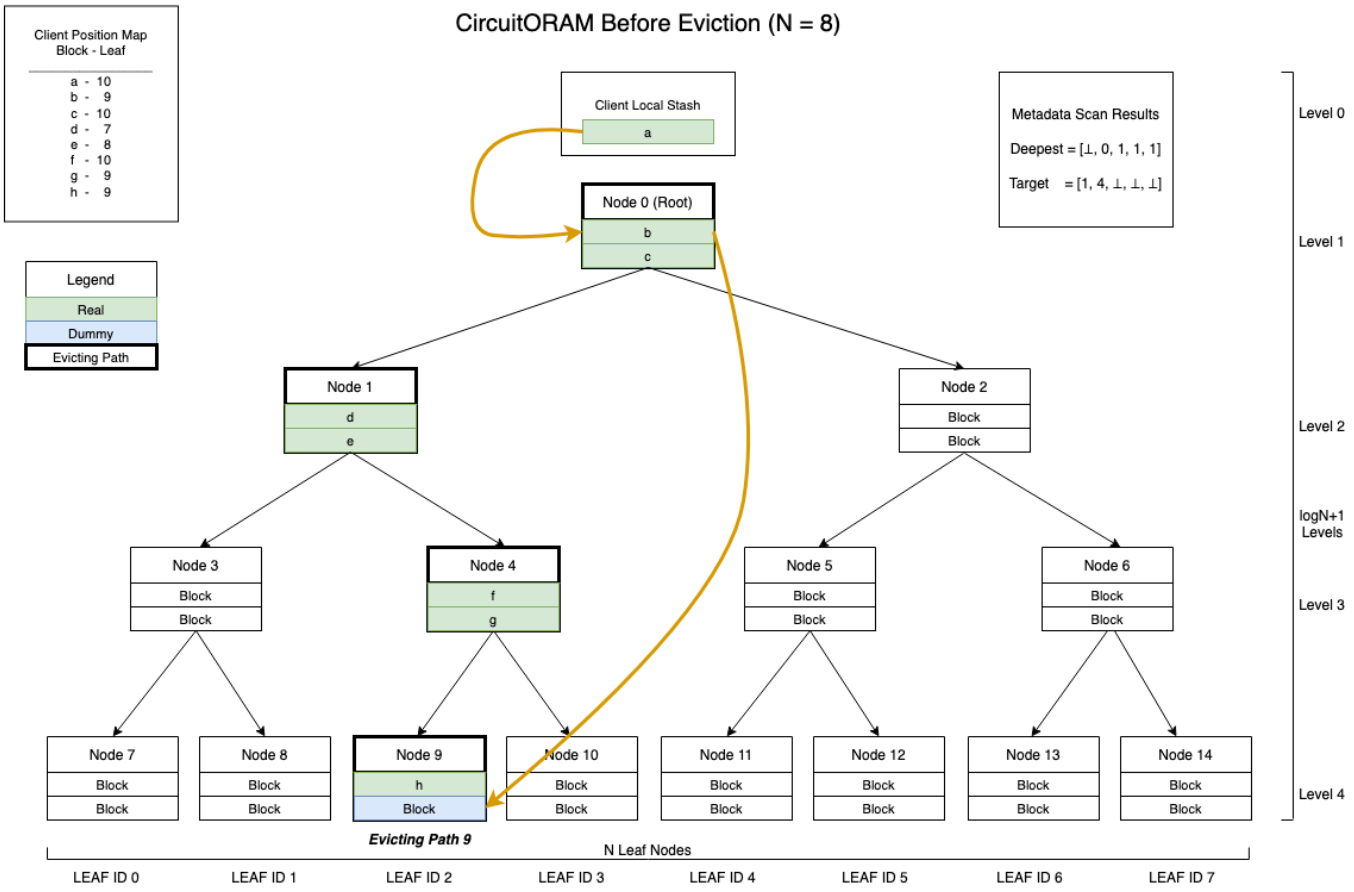

Fig. 8. An example of the CircuitORAM eviction process. Note that the client stash is referred to as Level 0 in CircuitORAM. One path from both the left and right sides of the tree is randomly chosen for eviction. For simplicity, this figure only shows one eviction path on the left side of the tree. The `Deepest` array represents the result from the first metadata scan, and the `Target` array represents the result from the second metadata scan. CircuitORAM's eviction process aims to prioritize evicting blocks from the shallowest levels of the tree (closest to the root) to the deepest levels of the tree (closest to the leaf). `Deepest[i]` represents the deepest level from which a block can be evicted. For instance, `Deepest[0] = ⊥`, as no block can be evicted to the stash at level 0. `Deepest[1] = 0`, as the block a from the stash (at level 0) can be stored in the root (at level 1). `Deepest[2] = 1`, as level 2 (Node 1) can successfully store a block from level 1(root node). The `Deepest` array and a second metadata scan create the `Target` array. The `Target` array specifies the target level that a block should be moved. For instance, `Target[0] = 1` specifies that a block from level 0 should be moved to level 1. `Target[2]= 1` means a block from level 1 should be moved to level 2. The rest of this `Target` array in this example is ⊥ as no block is being evicted from levels 2, 3, or 4. Given these metadata scan results, a single iteration of the eviction path results in `block a` being moved down to the root node and `block b` being moved down to the leaf level.

and eviction setup, and CircuitORAM's eviction process for a simple, efficient circuit would introduce a new ORAM construction that in theory, is both bandwidth and circuit efficient.

Finally, to introduce a new level of efficiency to these oblivious construction, concurrency would play a significant role in an improvement in performance. As seen in Figure 11, initialization of the tree on the server is expensive, but this simple task could be done concurrently. In addition to multi-threading the setup of the tree, tasks such as fetching nodes from the tree (on the server) and storing nodes back onto the tree (on the server) could be done in parallel. Figure 20 shows some preliminary results for how multi-threading could improve the performance of TreeORAM. For the experiment, fetching, storing, and searching through nodes locally were all parallelized using 12 threads. The most expensive, and more difficult task, such as evictions and tree setup, were not reconstructed to run in parallel. Furthermore, none of the code was properly profiled, so there are some expected inefficiencies in the TreeORAM implementation. Despite all these obstacles, Figure 20 shows a decent improvement in performance when using multi-threading for basic ORAM tasks. Scaling this to larger ORAM trees, with proper code profiling, and a fully concurrent implementation, would have a significant improvement in latency performance.

IX. Conclusion

This paper explains in detail the TreeORAM, PathORAM, CircuitORAM, and RingORAM constructions. Also provided are open source implementations to all these constructions as well as experimental results. Key observations include the expensive setup cost of ORAMs, the proportional expense of

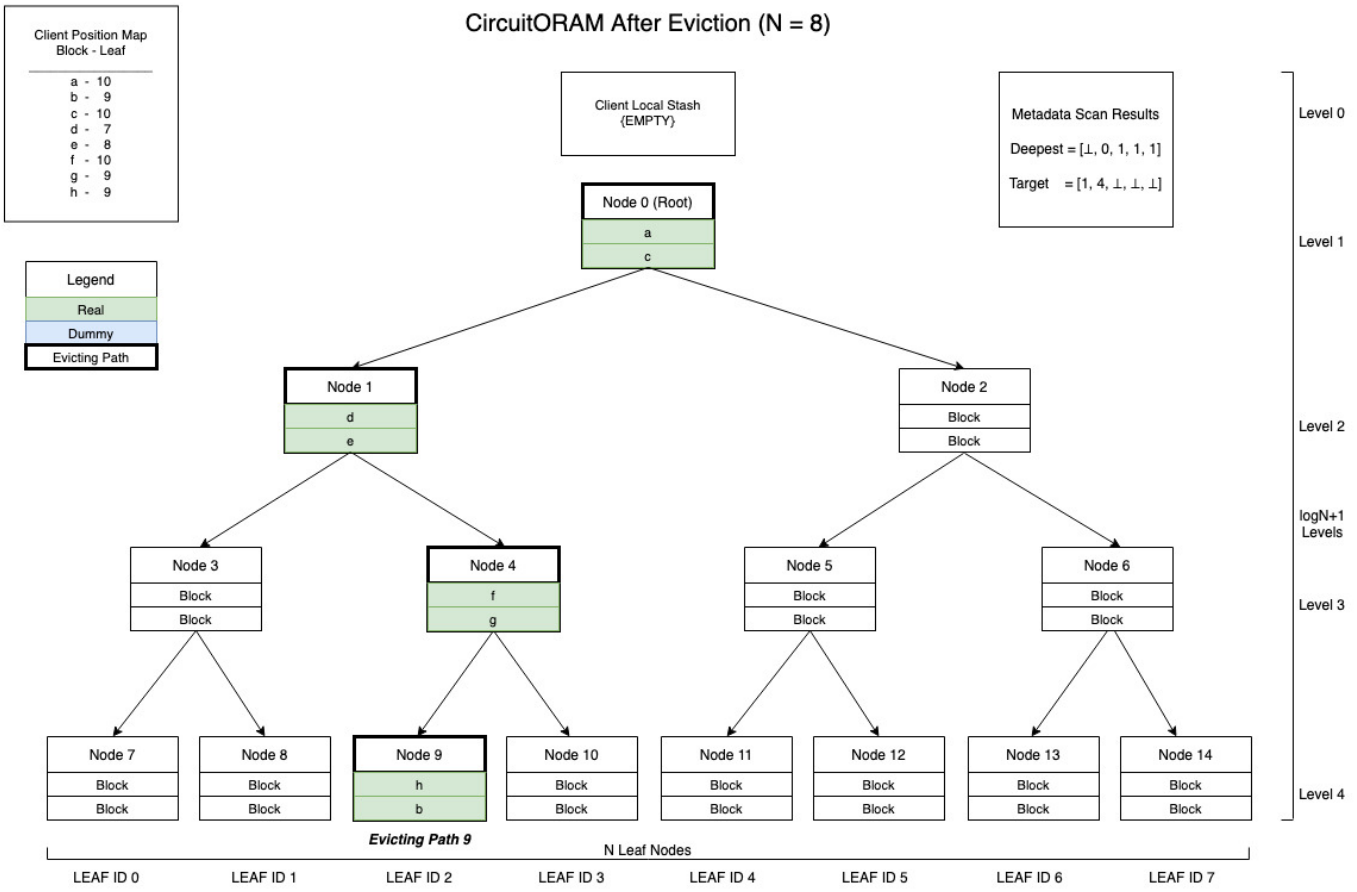

Fig. 9. The result of the CircuitORAM tree after performing an eviction on Path 9.

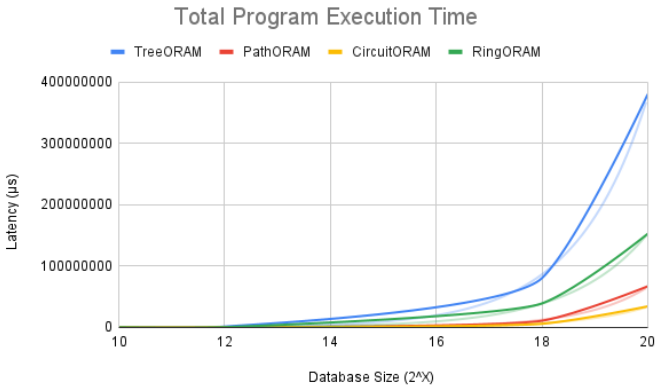

Fig. 10. The total execution time of each ORAM. Execution time is measured in microseconds. The darker lines represent the collected data. The lighter colored lines represent the exponential trend line for each ORAM. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

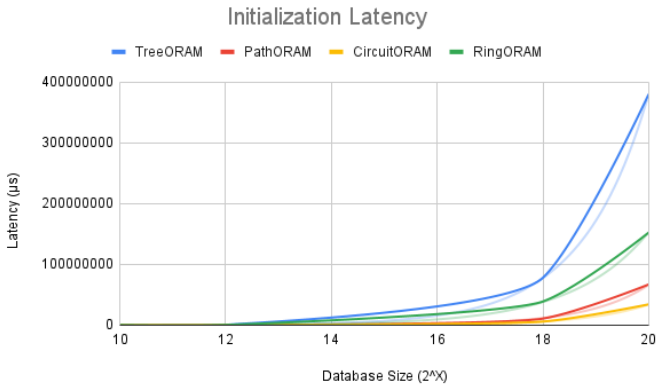

Fig. 11. The total initialization time of each ORAM. Latency is measured in microseconds. Initially, an ORAM tree is created, encrypted, and sent the the server. The darker lines represent the collected data. The lighter colored lines represent the exponential trend line for each ORAM. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

the eviction process in relation to the cost of access, and TreeORAM's inefficiencies due to its ever growing bucket size. A large amount of work is yet to be done to improve oblivious RAMs. Future work for improving oblivious RAMs include a further detailed analysis of the setup and eviction process, combining oblivious RAMs for new constructions, and introducing concurrency to oblivious RAMs to improve the latency of the aforementioned constructions.

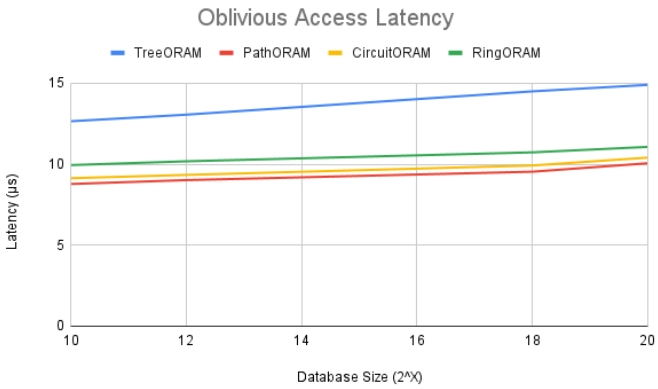

Fig. 12. The access time of each ORAM averaged over 100 data accesses. Access time is measured in microseconds. This graph shows a logarithmic scale for the access times. As can be seen in the figure, TreeORAM's data access process is significantly more expensive than PathORAM, CircuitORAM, and RingORAM. Figure 13 shows TreeORAM's accesses are hindered by the inefficient eviction process. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

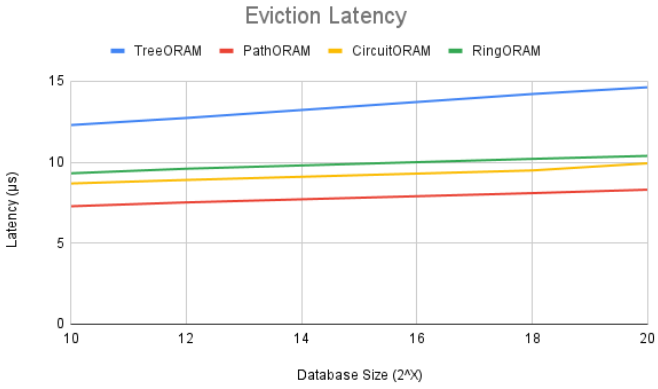

Fig. 13. The eviction time of each ORAM averaged over 100 data accesses. Execution time is measured in microseconds. This graph shows a logarithmic scale for the access times. As can be seen in the figure, TreeORAM's eviction process is significantly more expensive than PathORAM, CircuitORAM, and RingORAM. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

X. Acknowledgment

I want to thank Professor Ioannis Demertzis for being such a great project chair. Over the last two quarters, I learned a lot about the space of oblivious computing and had a great time working on this project. I also want to thank Professor Alvaro Cardenas for agreeing to be the Project Reader and for his meaningful and constructive feedback that made this paper better.

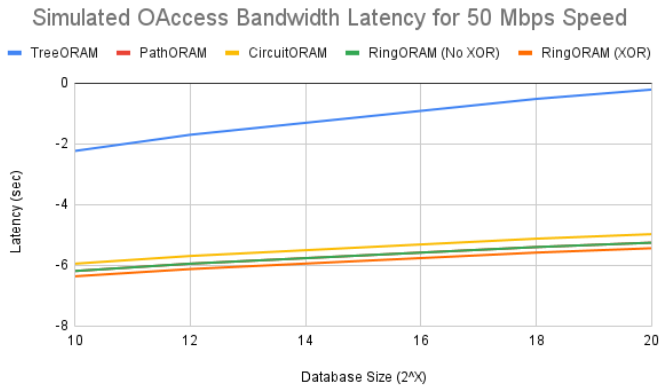

Fig. 14. The estimated bandwidth cost for a single data access calculated based on C++ implementations of these ORAMs. Data blocks are set to a size of 1KB. The simulated upload/download speed is 50 megabits/second. Latency is measured in seconds. This graph shows a logarithmic scale for the access times. Note that RingORAM (No XOR) and PathORAM have the same bandwidth cost for accesses but different costs for evictions. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

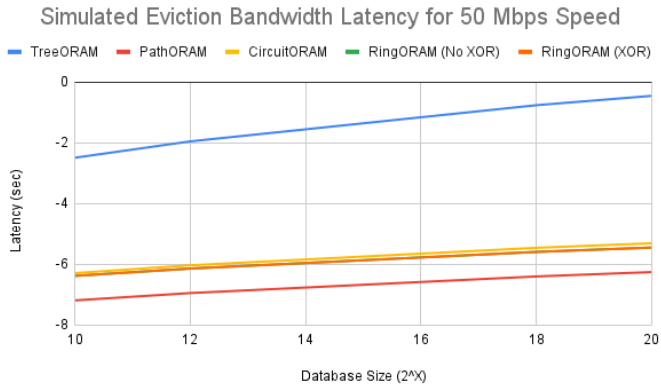

Fig. 15. The estimated bandwidth cost for a single eviction calculated based on C++ implementations of these ORAMs. Data blocks are set to a size of 1KB. The simulated upload/download speed is 50 megabits/second. Latency is measured in seconds. This graph shows a logarithmic scale for the access times. Note that RingORAM (No XOR) and RingORAM (XOR) have the different bandwidth cost for accesses but the same costs for evictions. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

References

[1] "COVID-19 Data Explorer," Our World in Data. [Online]. Available: https://ourworldindata.org/explorers/coronavirus-data-explorer. [Accessed: 27-May-2022].

[2] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In CCS, 2015.

[3] E. Shi, T.-H. Chan, E. Stefanov, and M. Li, "Oblivious Ram with O((logN)3) worst-case cost," Lecture Notes in Computer Science, pp. 197–214, 2011.

[4] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path Oram," Proceedings of the 2013 ACM SIGSAC conference on Computer amp; communications security - CCS '13, 2013.

[5] "Global Cloud Services Market Q1 2021," Canalys Newsroom, 29-Apr-2020. [Online]. Available: https://www.canalys.com/newsroom/global-cloud-market-Q121. [Accessed: 26-May-2022].

[6] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage," 29th USENIX Security Symposium, Aug. 2020.

[7] Islam, Mohammad Saiful et al. "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation." NDSS (2012).

[8] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S.

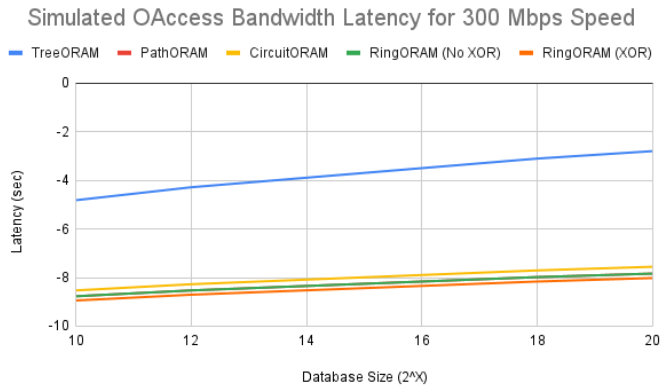

Fig. 16. The estimated bandwidth cost for a single data access calculated based on C++ implementations of these ORAMs. Data blocks are set to a size of 1KB. The simulated upload/download speed is 300 megabits/second. Latency is measured in seconds. This graph shows a logarithmic scale for the access times. Note that RingORAM (No XOR) and PathORAM have the same bandwidth cost for accesses but different costs for evictions. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

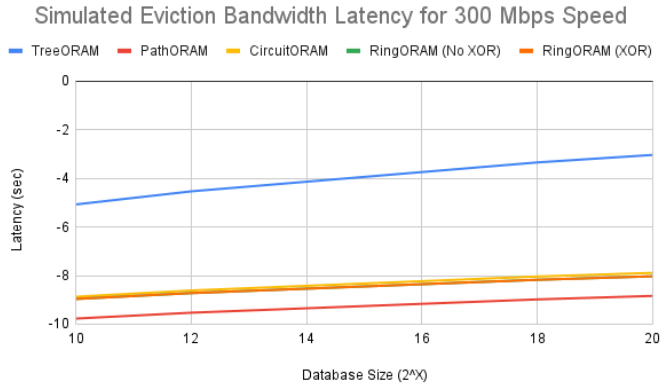

Fig. 17. The estimated bandwidth cost for a single eviction calculated based on C++ implementations of these ORAMs. Data blocks are set to a size of 1KB. The simulated upload/download speed is 300 megabits/second. Latency is measured in seconds. This graph shows a logarithmic scale for the access times. Note that RingORAM (No XOR) and RingORAM (XOR) have the different bandwidth cost for accesses but the same costs for evictions. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

Devadas, Constants Count: Practical Improvements to Oblivious RAM, pp. 415–430, Aug. 2015.

[9] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In Proceedings of the 4th ACM conference on Data and application security and privacy, pages 235–246. ACM, 2014

[10] S. Keswani, “TreeORAM Repository,” GitHub, 16-Apr-2022. [Online]. Available: https://github.com/sukeswan/TreeORAM.

[11] X. Wang, H. Chan, and E. Shi, “Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound,” Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.

[12] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In USENIX 2016.

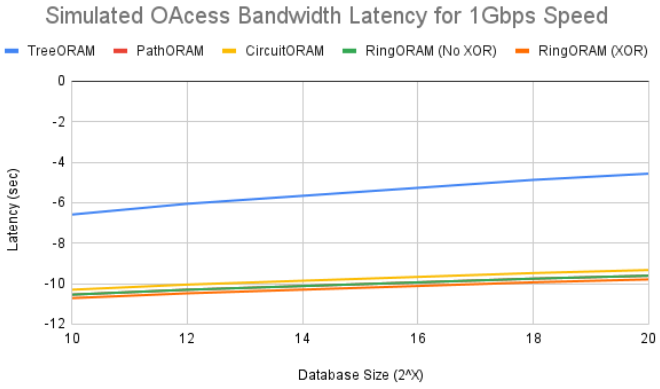

Fig. 18. The estimated bandwidth cost for a single data access calculated based on C++ implementations of these ORAMs. Data blocks are set to a size of 1KB. The simulated upload/download speed is 1 gigabit/second (1024 megabits/second). Latency is measured in seconds. This graph shows a logarithmic scale for the access times. Note that RingORAM (No XOR) and PathORAM have the same bandwidth cost for accesses but different costs for evictions. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

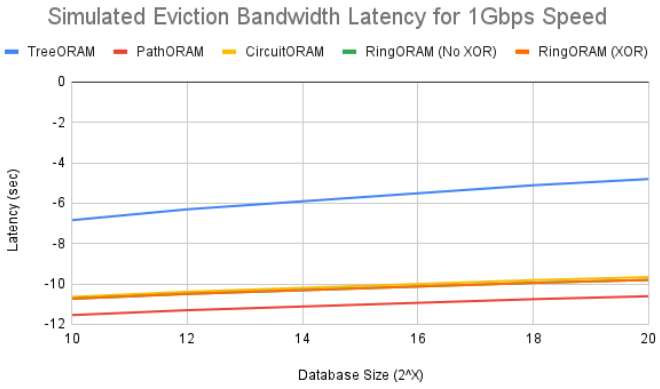

Fig. 19. The estimated bandwidth cost for a single eviction calculated based on C++ implementations of these ORAMs. Data blocks are set to a size of 1KB. The simulated upload/download speed is 1 gigabit/second (1024 megabits/second). Latency is measured in seconds. Latency is measured in seconds. This graph shows a logarithmic scale for the access times. Note that RingORAM (No XOR) and RingORAM (XOR) have the different bandwidth cost for accesses but the same costs for evictions. The X axis represents the size of the database. Example: 10 represents a database size of 2^{10}.

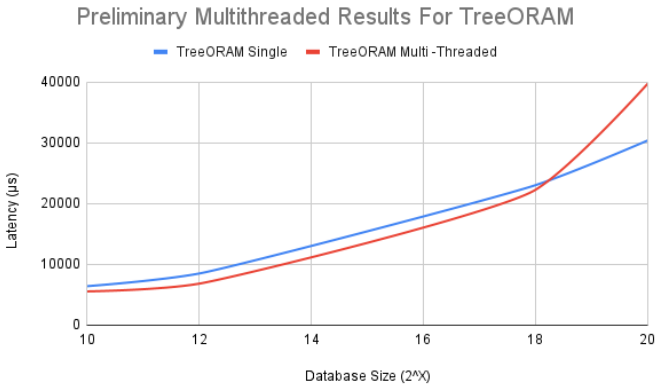

Fig. 20. Preliminary results of basic multi threading experiments with TreeORAM. This graph shows the data access latency averaged over 100 data accesses. Latency is measured in microseconds. We hypothesize that code inefficiencies hinder the performance of the multi-threaded TreeORAM implementation at a database size of 2^{18}, thus causing the worst performance for the multi-threaded version of TreeORAM.

	TreeORAM	PathORAM	CircuitORAM	RingORAM
Overall Runtime	$\mathcal{O}(\log^3 N)$	$\mathcal{O}(\log^3 N)$	$\mathcal{O}(\log^3 N)$	$\mathcal{O}(\log^3 N)$
Read Runtime	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$
Write Runtime	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$
Eviction Runtime	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log^2 N/A)$
Recursive Position Map Lookup Runtime	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Client Storage	$\mathcal{O}(1)$	$\mathcal{O}(\log N)\omega(1)$	$\mathcal{O}(\log N)\omega(1)$	$\mathcal{O}(\log N)\omega(1)$
Server Storage	$\mathcal{O}(N \log N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}((Z+S)N)$
Blocks per Bucket	$\log N$	4	2	$Z+S$
Online Bandwidth	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$

Fig. 21. The asymptotic costs broken down by category. Note for RingORAM, 3 parameters (Z, S, A) are chosen by the user.