A Practical Oblivious Cloud Storage System based on TEE and Client Gateway

Wensheng Zhang
Department of Computer Science, Iowa State University
Ames, Iowa, USA 50011
E-mail: wzhang@iastate.edu

2021 18th International Conference on Privacy, Security and Trust (PST) | 978-1-6654-0184-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/PST52912.2021.9647827

***Abstract*—In this paper, we propose a new oblivious cloud storage system, which is more efficient and scalable than existing schemes due to the combined leverage of SGX-based trusted execution environment (TEE) at the cloud server side and the moderate storage space at the client side. The TEE is employed to securely implement functionalities of ORAM model in the server without tightly involving the clients. Meanwhile, the storage at the client side is utilized to store metadata and recently/frequently accessed data, which facilitates the client to remotely determine the strategies for data query/eviction and to reduce the frequency of directly accessing data from the server. The evaluation results show that, when the size of outsourced data is 1-20 GB and the block size is 1-8 KB, the data access throughput between 320 KB/s and 640 KB/s can be attained, and the average query latency for each block is only 2.26-12.80 ms.**

I. Introduction

Clients of cloud storage service may worry that the content and access patterns to their outsourced data could be revealed to the service provider, its employees, or attackers invading the system. Encryption algorithms can protect the secrecy of data content, and a variety of oblivious RAM (ORAM) [1] based techniques have been developed to protect the clients' data access pattern. Though last decade has witnessed the significant improvement in the efficiency of the ORAM-based techniques, there is still a noticeable performance gap between an ORAM-based oblivious storage system and a normal storage system without obliviousness in data access.

In this paper, we propose a new oblivious cloud storage system that further improves the efficiency of ORAM constructions by leveraging two recent developments in cloud system architecture: the emerging trusted execution environments (TEEs) enabled by more accessible technologies such as Intel Software Guard Extension (SGX) [2] and Arm's TrustZone [3]; the increasing popularity of hybrid cloud model.

TEEs can be utilized to securely implement some functionalities of ORAM model in the server without tightly involving clients. With the hybrid cloud model, a client can run a local gateway with moderate storage resources. To achieve efficiency and scalability, the gateway outsources most of the client's data to a remote cloud storage server, while retaining meta and recently/frequently-accessed data to reduce the frequency of accessing data directly from the remote server. There have been efforts [4], [5], [6] on implementing ORAM models in SGX enclaves and efforts [7] on improving the efficiency of ORAM construction based on the hybrid cloud model. However, to the best of our knowledge, there has no prior effort in leveraging both of them.

Our proposed system is based on the ORAM construction proposed by Ma and Zhang [7] but extend it significantly in both the cloud server and the client sides. In the server side, a trusted component is executed in an Intel SGX enclave to obliviously retrieve the query target block from a sequence of *randomly* selected blocks; it also obliviously permutes and re-encrypts data blocks on a path to implement the eviction (shuffling) function of ORAM. The following techniques are used to make the oblivious retrieval and eviction more efficient and accountable. First, each data block is doubly encrypted, where authenticated cipher AES-GCM is used to encrypt the plaintext for data confidentiality and integrity, and more computationally-lightweight SHA256-XOR cipher is used to efficiently hide the data permutation pattern during the eviction process. Second, data blocks are permuted and re-encrypted piece by piece in the enclave memory to further improve the efficiency and scalability of eviction. Third, reading/writing data blocks from/to storage to/from memory is conducted in parallel with the computationally-intensive permutation/re-encryption process in enclave, which also reduces the latency of eviction. In the client side, data structures are thoughtfully-designed to keep recently/frequently accessed data in the client gateway while gradually evicting the least-frequently accessed data to the server. The client side design also enables parallelism of data query and eviction processes, and hence reduces the data query latency experienced by the client.

A prototype of the proposed design is implemented to evaluate the performance. The result shows that, when the size of outsourced data is 1-20 GB and the block size is 1-8 KB, the attained data access throughput is between 320 KB/s and 640 KB/s, and the average query latency for each block is only 2.26-12.80 ms. The performance is significantly better than that incurred in state of the art systems.

In the rest of the paper, Section II defines the research problem. Section III describes our proposed design. Section IV presents performance evaluation. Section V briefly summarizes related works. Finally, Section VI concludes the paper.

II. Problem Definitions

A. System Model

We consider a system composed of a cloud storage server (called *server* hereafter), a client-side on-premise storage gate-

way (called *client gateway* hereafter), and a set of clients who access data via the client gateway. The client gateway has a moderate storage capacity, though is still much smaller than (say, around 0.1% of) the server's.

Assume there are up to N data blocks accessible to the clients. The client gateway may store a small subset of the blocks accessed recently/frequently, but export the rest to the cloud server. Each *data access* from the client gateway to the cloud storage is of two types: read a data block D of unique ID i from the cloud server's storage; write a data block D of unique ID i to the cloud server's storage. The client gateway should encrypt the data blocks and hide the block IDs from the server, and should store the keys used. As the content and IDs of the accessed data blocks are both hidden, the cloud server (as well as any one who can observe the cloud server's behavior) can only observe the following two types of access to storage locations: retrieve (i.e., read) a data block D from location l at the cloud server's storage; upload (i.e., write) a data block D to location l at the cloud server's storage.

B. Security Assumptions and Design Goals

We aim to protect the privacy of data access pattern for the clients against the cloud storage service provider, its employees, or attacker who has invaded the cloud storage system, who altogether are referred to as the *adversary*.

With the trusted execution environment (TEE) technology such as Intel Software Guard Extension (SGX), the cloud server can run in the *trusted* mode or the *un-trusted* mode. The server's main memory includes the *trusted* enclave space, where the data is encrypted, and the *un-trusted* regular space. The enclave space is much smaller than regular space; particularly, the total size of enclave space is usually only 64 or 128 mega bytes, which is divided into pages each of 4 kilobytes.

When the server runs in the trusted mode, it can only execute code in enclave, but can access data both inside or outside enclave. We assume that, in this mode, the pattern of the server's accesses within a page of the enclave memory space (including what data are accessed, the types of access, and in which order are the accesses) is not observable by the adversary [5]. Hence, we aim to protect the access privacy at the granularity of page.

We assume the clients and the client gateway are trusted and not compromised, and thus their behaviors (i.e., behaviors occurring within the clients' on-premise environment) cannot be observed by the adversary.

Our design goals are twofold: security and efficiency. As our design is based on the ORAM proposed by Ma and Zhang, for which the security (i.e., privacy preservation) has been proved, in this paper we mainly target at achieving page-level privacy in accessing the enclave memory and improving the efficiency which is measured by the latency for data query and eviction.

III. Proposed Design

This section presents our proposed design, in terms of data block encryption, storage organization, query and eviction.

A. Double Encryption of Data Blocks

Each data block is encrypted twice for dual purposes: protecting data secrecy and integrity; making decryption/re-encryption more efficient during the eviction process.

Let k denote the secret key used in the double encryption. Each data block whose plain text is denoted as b, is first encrypted and authenticated using k and an authenticated encryption algorithm such as AES-GCM (referred to as GCM hereafter). The result is denoted as b'. Then, b' is divided into multiple 256-bit (i.e., 32-byte) pieces denoted as $(b'[1], b'[2], \cdots)$. Next, these pieces are encrypted using k, secure hash function $SHA256$ and the bit-wise XOR operation $\oplus$ as follows: a 32-byte vector $b''[0]$ is randomly generated; each $b'[i]$ for $i = 1, 2, \cdots$, is transformed to $b''[i] = b'[i] \oplus SHA256(k|b''[0]|i)$. The resulting $(b''[0], b''[1], b''[2], \cdots)$ becomes the ciphertext of b after double-encryption. Decryption is the reverse of the above encryption procedure.

B. Storage Organization at Cloud Server

In the server, the space for storing encrypted data blocks is organized as a m-ary storage tree, the same as in [7]. Following the work, we also use L to denote the height of the tree, Z_0 the capacity of each leaf node and Z_1 the capacity of each non-leaf node. Different from [7], each block stored in the tree should have been doubly-encrypted as above.

The enclave memory keeps the secret key k, which is shared with the client gateway, and also allocates several pages to facilitate oblivious permutation and decryption/re-encryption of data blocks. The details about these pages are discussed in Section III-E. Additionally, the cloud server also maintains in the un-trusted memory space several buffers. The details of these buffers are discussed in Section III-D.

C. Storage Organization at Client Gateway

The client gateway maintains in its main memory an *index table* for all of the N real data blocks and an *index block* for each node on the cloud server's storage tree. In addition, it maintains the following data structures to facilitate concurrent query and eviction: R, which is a ring that can store up to $2q$ blocks, where q is the number of queries before a round of eviction process can be launched; Q, which a queue of query requests waiting to be sent.

Ring R has pointers p_0, p_1 and p_2. Pointer p_0 points to the first of the q blocks that are being evicted at the server for the current eviction round, p_1 points to the first of the blocks that will be evicted at the server in the next eviction round, and p_2 points to the space for the block that will be next uploaded to the server for eviction. Hence, the blocks from the position pointed to by p_1 to the position ahead of that pointed to by p_2 will be evicted in the next eviction round. Note that, the blocks from the position pointed to by p_0 to the position before p_2 are replicas of blocks at the server. By keeping these replicas, clients are allowed to query the blocks that are currently being evicted at the server without accessing them from the server; this way, eviction can be conducted without being interfered with by ongoing queries.

Queue Q queues up pending query requests, where each query is either retrieval-type or update-type. A retrieval-type query contains only a block ID of the query target, and aims to retrieve the content of the desired block from the server. An update-type query contains both a block ID and up-to-date block content, and aims to retrieve the desired block and then replace it with the up-to-date content.

D. Data Block Query

Data block query is accomplished through the collaboration among the client gateway, the un-trusted component of the server, and the trusted enclave of the server.

The process is initiated by the client gateway, who first looks up its index table to find out the path ID, denoted as p_t, of the query target block with ID t. Then, the client gateway selects the blocks that should be accessed from the server's storage tree following the query algorithm proposed in [7]. The location sequence of the selected blocks can be denoted as $\langle (l_0,\ o_0),\ (l_1,\ o_1),\ \cdots,\ (l_{y-1},\ o_{y-1})\rangle$, where for each $i \in \{0, \cdots, y-1\}$, pair (l_i, o_i) indicates that the block with intra-layer offset o_i on layer l_i should be accessed from the server's storage tree. Among the pairs, assuming the query target block's location is indicated by pair (l_x, o_x), the client gateway encrypts index x to ciphertext $x' = enc_{GCM,k}(x)$ using the shared secret key k and authenticated encryption algorithm GCM. Then, the client gateway sends the following query request to the un-trusted component of the server: $(\langle (l_0,\ o_0),\ (l_1,\ o_1),\ \cdots,\ (l_{y-1},\ o_{y-1})\rangle, x')$.

Upon receiving the query request, the un-trusted component of the server reads the data blocks whose locations are indicated by the pairs into a temporary buffer buf in main memory. Then, it calls the query function in the enclave, named $enclave\_query$, with arguments: the pointer to buffer buf, the size of each block, and x' received from the client gateway. The query function works as follows to securely and obliviously extract the query target block, re-encrypt it, and return the block in ciphertext.

- The function allocates within the enclave a temporary buffer, denoted as buf', which is large enough to store a data block, and then x' is decrypted to x.
- The block with index x is obliviously read into the enclave temporary buffer. For this purpose, the following operation is conducted for each of the y blocks: for each piece i of the block, it is loaded to a register R_0; piece i of buf' is loaded to another register R_1; if the block is the query target, content of R_0 is written to piece i of buf', or otherwise, content of R_1 is written to the piece (i.e., the piece is unchanged).
- Block buf' is doubly-decrypted. If the GCM-based decryption fails, the client gateway reports that the block's integrity has been compromised and then aborts. Otherwise, the block is re-encrypted with GCM and key k.

E. Data Block Eviction

Data block eviction is also accomplished through the collaborations between the client gateway, and the un-trusted and the trusted (enclave) components of the server.

1) Client-side Operations: The process is started by the client gateway. From the blocks that it stores, the client gateway uploads those the least recently accessed to the un-trusted component of the server, who stores the received blocks into the buffer in the main memory. Note that, due to the limits in the memory space and the speed of data block eviction with the server, the client gateway cannot upload blocks too fast.

After every q blocks have been uploaded, a new round of eviction can be started. Also, the client gateway needs to figure out how the q blocks should be evicted in the cloud server by running the eviction algorithm proposed in [7], and this results in the following sequence of L permutation maps (recall that L is the height of the storage tree): $\Pi = \langle\ \pi_0,\ \pi_1,\ \cdots,\ \pi_{L-1}\ \rangle$, and each π_i is a permutation of $\{0,\ 1,\ \cdots,\ s_i - 1\}$, where $s_i = q + Z_1$ for $i < L-1$ (i.e., layer i is the leaf layer) and $s_i = q + Z_0$ for $i = L-1$ (i.e., layer i is a non-leaf layer). Here, each π_i in the sequence should be used as follows: When the eviction is conducted on layer i, the ordered sequence of blocks stored in the evicting node on this layer is appended by the ordered sequence of current evicting blocks to form a combined ordered sequence of s_i blocks. This combined sequence is then re-ordered according to permutation π_i; that is, the block on position j in the combined sequence should be moved to position $\pi_i[j]$. Then, the first Z_1 (if $i < L-1$) or Z_0 (if $i = L-1$) blocks of the re-ordered sequence should be moved to the evicting node on this layer, while the rest q blocks are dropped if $i = L-1$ or otherwise become the new evicting blocks to be used during the eviction for layer $i+1$.

The above permutation maps should not be exposed to the un-trusted component of the server; otherwise, the adversary can track how the blocks are shuffled. As the client gateway cannot directly communicate remotely with the program in the trusted enclave, the permutations should be encrypted and also carry authentication code to protect its confidentiality and integrity. For this sake, the client uses the GCM algorithm to obtain an authenticated encryption of each π_i, denoted as $\tilde{\pi}_i = enc_{GCM,k}(\pi_i, timestamp)$, and sends $\tilde{\Pi} = \langle \tilde{\pi}_i, \cdots, \tilde{\pi}_{L-1}\rangle$ to the untrusted component of the server. Note that, timestamp is included to prevent replay attacks.

2) Server-side Operations: Upon obtaining and decrypting the permutation maps, in theory, the enclave component could conduct data block eviction directly on behalf of the client gateway. However, this is infeasible because the enclave has too limited memory space to contain the blocks needed for secret permutation. To address this limitation, each round of eviction is conducted layer by layer, and for each layer, we adopt *piece-wise eviction* that works as follows.

Operations by the Un-trusted Component. When the eviction is conducted for layer i, where $i = 0, \cdots, L-1$, of an eviction round, the un-trusted component allocates the following buffer spaces for the trusted server to access the information too large to fit in the enclave memory space:

- $buf_{\tilde{\Pi}}$ stores the sequence $\tilde{\Pi}$ of encrypted permutation maps used for the current round of eviction.

- $buf_{\tilde{\Pi}'}$ stores the sequence of encrypted permutation maps to be used for the next round of eviction.
- buf_{CEB} stores the current evicting blocks (CEBs) for layer i of the current eviction round.
- buf_{CEN} stores the blocks in the current evicting node (CEN) of the current evicting round; i.e., the node on layer i of the current eviction path.
- buf_{NEB} stores the next-round evicting blocks (NEBs) to be used for the next round of eviction round.
- buf_{NEN} stores the blocks in the next evicting node (NEN).
- buf_{WB} stores the blocks that needed to be written back to the storage tree at the secondary storage.

Here, each of the above buffer has the size of q blocks.

The un-trusted server is responsible for filling/dumping information to/from the above buffers, and the processes for filling/dumping should be conducted as concurrently as possible for system performance. Initially, all these buffers are empty. Then, the following rules are applied to fill/dump the buffers concurrently:

- Whenever a block is uploaded from the client gateway, the block is appended to the tail of buf_{NEB}.
- When a new sequence of encrypted permutation maps is received, the sequence is saved to $buf_{\tilde{\Pi}'}$.
- When NEB_buf is empty, the un-trusted server proactively uploads the blocks in the next evicting node to NEB_buf. Note that, if an eviction process is ongoing, the next evicting node is obviously the child of the current evicting node on the current evicting path; if there is no eviction process ongoing, the next evicting node is the root of the storage tree.

When $buf_{\tilde{\Pi}'}$, buf_{NEB} and buf_{NEN} have been filled, and meanwhile there is no eviction process ongoing, the un-trusted component does the following to prepare for a new round of eviction:

- It swaps $buf_{\tilde{\Pi}}$ with $buf_{\tilde{\Pi}'}$, buf_{CEN} with buf_{NEN}, and buf_{CEB} with buf_{NEB}.
- It identifies the current evicting path (CEP), based on the deterministic reverse-lexicographic order for selecting evicting paths.

Then, it calls the enclave's $evict$ function, with $\tilde{\Pi}_0$ stroed in $buf_{\tilde{\Pi}}$, buf_{CEN} and buf_{CEB} as parameters, to start eviction for layer 0 for the current evicting path CEP.

When the evict function completes for layer i of the current evicting path, the un-trusted component is informed and responds as follows:

- It waits for buf_{WB} to be empty. Then, it swaps buf_{WB} with buf_{CEN} and starts a thread to write the blocks in buf_{WB} back to the current evicting node.
- If layer i is the leaf layer, it will empty $buf_{\tilde{\Pi}}$ and buf_{CEN}, wait for $buf_{\tilde{\Pi}'}$, buf_{NEN} and buf_{NEB} to be filled, and then start a new eviction process as described above.
- If layer i is not the leaf layer, it will wait for buf_{NEN} to be filled; then, it will swap buf_{CEN} with buf_{NEN}, and call enclave's $evict$ function to evict blocks for layer $i+1$ with the parameters including $\tilde{\pi}_{i+1}$ stored in $buf_{\tilde{\Pi}}$, buf_{CEN} and buf_{CEB}.

Operations by the Trusted Component (Enclave). The trusted enclave exposes a function named *evict*, which can be called by the un-trusted component to conduct eviction for layer i of the storage tree. Essentially, the function is to obliviously permute and re-encrypt a set of blocks, denoted as buf', which is the ordered sequence including the blocks in the current evicting node (CEN) in buf_{CEN} and the current evicting blocks (CEBs) in buf_{CEB}. Here, the number of blocks in buf' is no greater than $\max(Z_0, Z_1) + q$; recall that Z_0 and Z_1 are the number of blocks in a leaf and non-leaf node, respectively. In the practical settings, the number of blocks in buf' is on the order of thousands but usually no greater than 8 thousands. Given the limited size of trusted memory space, it is not feasible to obliviously permute/re-encrypt the blocks in the unit of block; hence, we propose to process the blocks in a small unit of 32-byte piece.

Let buf' denote the concatenation of the sequences of blocks in buf_{CEN} and buf_{CEB} i.e., $buf' = buf_{CEN}|buf_{CEB}$, and $|buf'|$ denote the number of blocks in buf'. To support oblivious piece-wise permutation/re-encryption of these blocks, the trusted server allocates the following smaller data structures to temporarily buffer the secret information needed for the processing:

- π_i: permutation map for blocks in buf'.
- $ORVP$: 64 pages (each of $4K$ bytes) store the old random vectors of the blocks; each page j of $ORVP$, denoted as $ORVP[j]$, stores $|buf'|$ words where each word has 4 bits; hence, $ORVP[j][b]$ stores word j of the old random vector for block b in buf'.
- $NRVP$: similar to $ORVP$, $NRVP$ has 64 pages storing the new random vectors of the blocks; each page $NRVP[j]$ stores $|buf'|$ 4-bit words; hence, $NRVP[j][b]$ stores word j of the new random vector for block b in buf'.
- $PIECE$: similar to $ORVP$ and $NRVP$, $piece$ has 64 pages each storing $|buf'|$ 4-bit words; hence, each $PIECE[j][b]$ stores word j of a 32-byte piece of block b in buf'.
- k: the key shared between the enclave and the client.

The evict function works as follows. In the beginning, it decrypts the encrypted permutation map $\tilde{\pi}_i$ passed in by the untrusted component, to obtain permutation map π_i in plain text. Then, it loads into the random vector pages $ORVP$ the pieces with index 0 of every encrypted blocks in buf'. Each piece is divided into 64 words each of 4 bits, and each word $p \in \{0, \cdots, 63\}$ is stored to page p of $ORVP$ respectively; this way, no matter which block's old random vector is accessed, all the pages of $ORVP$ have to be accessed, which achieves the page-level obliviousness. Note that, each encrypted data block is treated as a sequence of 32-byte pieces, and thus each block has $block\_size/32bytes$ pieces; piece 0 is the random vector used in the SHA256-XOR

encryption of the block and the rest blocks are the ciphertexts of the encryption. In addition, the evict function also initializes the new random vectors by filling randomly-generated data into the new random vector pages $NRVP$, and stores these random vectors to every block as their piece 0.

After the above completes, the evict function re-encrypts and permutes the blocks in buf' piece-by-piece. Specifically, for each piece $j = 1, \cdots, block\_size/32bytes - 1$, the following steps are applied on each block with index b in buf': First, piece j of block b, is loaded into a temporary variable $piece$. Then, the piece is decrypted by bitwise-XORed with

$$v = SHA256(ORVP[0][b], \cdots, ORVP[63][b]),$$

marked to be moved to index $b' = \pi_i[b]$ of buf', and re-encrypted by bitwise-XORed with

$$v' = SHA256(ORVP[0][b'], \cdots, ORVP[63][b']).$$

Following the above steps, the resulting piece is divided into 4-bit words, and each word $p \in \{0, \cdots, 63\}$ is stored to word b' in page p of $PIECE$; note that, all of the pages of $PIECE$ are accessed regardless of the permutation map, which also achieves page-level obliviousness. After the above steps have been applied over all the blocks in buf', the resulting pages $PIECE$ are written back to buf' as piece j of the blocks.

IV. Performance Evaluation

We evaluate our designed scheme on a computer with Intel Core i5-8400 CPU (2.80GHz) of six cores and a RAM of 8.00GB. Both the cloud storage server (including the trusted and un-trusted components) and the client gateway are run on the same computer. This way, the impact of network communication delay, which varies significantly in different settings is ignored. The impact of network bandwidth is not considered because the designed scheme only incurs the communication overhead of sending query/eviction strategy, which is very small compared to the amount of data accessed/shuffled. A client is simulated, which runs the following loop to continuously generate query requests: it randomly generates a block ID; requests the client gateway to query the target block; waits for the result; and then randomly generates the next block ID to query. Similar to [7], we set security parameter to 32, and thus one round of eviction is launched after every $25\lambda = 800$ queries have been processed; We set parameter $m = 8$ as it balances the performance and storage cost. The proposed design is evaluated on the following metrics:

- Query latency (best), which is measured as the average time to complete a query in the best scenario (i.e., the query can be immediately handled without any ongoing eviction or in parallel with an ongoing eviction).
- Query latency (avg.), which is measured as the average time to complete a query (including a query that can be processed immediately and that having to wait).
- System throughput, which is measured as the amount of data (in the unit of bytes) that can be queried per second.

In the following, Tables I, II and III show the main results of evaluation, where each number is an average over the time period that around 10,000 queries are processed.

data size	query latency (best)	query latency (avg.)	throughput
1 GB	0.54 ms	2.26 ms	453.81 KB/s
2 GB	0.63 ms	2.59 ms	395.13 KB/s
4 GB	0.66 ms	2.90 ms	353.59 KB/s
10 GB	0.86 ms	3.14 ms	325.98 KB/s
20 GB	0.88 ms	3.20 ms	319.20 KB/s

Table I: Performance with varying data size and 1 KB block.

Table I shows the performance when the size of data block is fixed at 1 KB and the size of the total outsourced real data varies from 1 GB to 20 GB. As can be observed, along with the increasing data size, the query latency increases and the data access throughput decreases. This is because, as the data size increases, the size and height of the storage tree increases as well, which results in higher cost for query and eviction. Meanwhile, we can find that the increase in latency and the decrease in throughput are much slower than the increase of data size. For instance, when the data size increases from 1 GB to 2 GB, the average query latency increases from 2.26 ms to 2.59 ms and the throughput drops from 453.81 KB/s to 395.13 KB/s; however, when the data size increases from 10 GB to 20 GB, the average query latency only slightly increases from 3.14 ms to 3.28 ms and the decrease of throughput from 325.98 KB/s to 319.20 KB/s is small as well. Hence, the result indicates the scalability of our design in term of the delivered performance contrast to the size of outsourced data.

block size	query latency (best)	query latency (avg.)	throughput
1 KB	0.86 ms	3.14 ms	325.98 KB/s
2 KB	0.77 ms	4.96 ms	413.31 KB/s
4 KB	0.67 ms	6.93 ms	590.99 KB/s
8 KB	0.82 ms	12.16 ms	673.46 KB/s

Table II: Performance with varying block size and 10 GB data.

Table II shows the performance when the size of outsourced real data is fixed at 10 GB but the block size varies from 1 KB to 8 KB. As expected, the average query latency increases with the block size. However, the increase in the per-block query latency is slower than the increase of block size, which results in the increase of data access throughput. The similar trend can be observed in Table III, which shows the performance when the number of outsourced real data blocks is fixed at 2 million but the block size varies from 1 KB to 8 KB.

block size	query latency (best)	query latency (avg.)	throughput
1 KB	0.63 ms	2.59 ms	395.13 KB/s
2 KB	0.61 ms	4.02 ms	509.10 KB/s
4 KB	0.69 ms	6.90 ms	593.59 KB/s
8 KB	0.74 ms	12.80 ms	640.18 KB/s

Table III: Performance with varying block size and 2 M blocks.

Comparisons to the most related works. Among the works related to ORAM and SGX, the ZeroTrace system by Sasy et al. [4] and the SGX-based ORAM constructions by Rachid et

al. [5] are the most related. Our system includes both cloud storage server and client gateway, where the client gateway maintains information on how the outsourced data is stored in the cloud storage and instructs the trusted component at the server side to perform data query and eviction. However, the related systems [4], [5] implement the oblivious access completely in the server side. Also, our system adopts the piece-wise approach for data eviction, which is more efficient in supporting large block size. Due to the differences, our system has better performance. Particularly, when the size of the outsourced data is 10 GB and the block size if 1 KB, our system incurs the average query latency at 3.14 ms, but the ZeroTrace system has the latency of 49 ms and 140 ms when the underlying ORAM construction is Path ORAM and Circuit ORAM, respectively, and the data backend is HDD, and the constructions implemented by Rachid et al. [5] has even higher latency. We attribute the performance difference mainly to the difference in system settings, and hence do not conduct more detailed comparison in this paper.

V. Related Works

Many ORAM constructions have been proposed since Goldreich and Ostrovsky [1], [8] first introduced the concept. The constructions roughly fall into two categories, hash-based and index-based. The hash-based ORAMs [1], [9], [10], [11], [12], [13] organize the data storage as layers and use hash tables for data look up, while the index-based ORAMs [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24] rely on index tables. Clients of index-based ORAMs are required to access index tables, either stored locally or outsourced to the server (sometimes recursively). The construction proposed by Ma and Zhang [7], which is the underlying ORAM construction of our proposed system, is a hash-based ORAM. Representative index-based ORAMs include SSS ORAM [15], binary tree ORAM (T-ORAM) [14], Path ORAM [17], etc. Particularly, the Path ORAM is also a tree-based ORAM that organizes its data storage as a binary tree; each query and eviction operation requires a root-to-leaf path on the tree to be completely accessed. It is the underlying ORAM in [4] and [5].

Before the ORAM functionality was proposed to be implemented in TEE, improving the efficiency of client-server communication has been a focus of research. Several ORAMs have been developed to have constant client-server bandwidth-blowup. For example, Devadas et al. proposed Onion-ORAM [22] and Moataz et al. proposed C-ORAM [18]. Hoang et al. [25] proposed S^3ORAM based on the deployment of multiple (at least three) non-colluding servers, to achieves $O(1)$ bandwidth-blowup for client-server communication at the cost of requiring $O(\log N)$ bandwidth-blowup for communication between the servers.

VI. Conclusion and Future Work

This paper proposes a practical oblivious cloud storage system, based on the combined leverage of SGX-based trusted execution environment (TEE) at the cloud server side and the moderate storage space at the client side. The evaluation results show that, when the size of outsourced data is 1-20 GB and the block size is 1-8 KB, the data access throughput between 320 KB/s and 640 KB/s can be attained, and the average query latency for each block is only 2.26-12.80 ms. In the future, we plan to further improve the performance of the system by applying techniques such as multi-thread execution of eviction inside the enclaves. We also plan to polish the implementation and make the system more robust to deploy.

Acknowledgement

The work is supported by NSF under grant CNS-1844591.

References

[1] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, 1996.

[2] V. Costan and S. Devadas, "Intelsgxexplained," *IACR Cryptology ePrint-Archive*, pp. 1–118, 2016.

[3] "Arm TrustZone Technology," https://developer.arm.com/ip-products/security-ip/trustzone, [Online; accessed 1-August-2021].

[4] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel SGX," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[5] M. H. Rachid, R. Riley, and Q. Malluhi, "Enclave-based oblivious ram using intel's sgx," 2020.

[6] T. Hoang, R. Behnia, and Y. Jang, "Mose : Practical mutli-user oblivious storage via secure enclaves," in *ACM CODASPY*, 2020.

[7] Q. Ma and W. Zhang, "Efficient and accountable oblivious cloud storage with three servers," *IEEE CNS*, 2019.

[8] O. Goldreich, "Towards a theory of software protection and simulationon oblivious rams," *Proc. SIGACT STOC*, 1987.

[9] M. T. Goodrich and M. Mitzenmacher, "Mapreduce parallel cuckoo hashing and oblivious ram simulations," *Proc. CoRR*, 2010.

[10] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Proc. CRYPTO*, 2010.

[11] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," *Proc. ICALP*, 2011.

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious ram simulation with efficient worst-case access overhead," *Proc. CCSW*, 2011.

[13] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proc. CCS*, 2012.

[14] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with $o((\log n)^3)$ worst-case cost," *Proc. ASIACRYPT*, 2011.

[15] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *Proc. NDSS*, 2011.

[16] C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, "Optimizing oram and using it efficiently for secure computation," *Proc. PETS*, 2013.

[17] E. Stefanov, M. V. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," *Proc. CCS*, 2013.

[18] T. Moataz, T. Mayberry, and E.-O. Blass, "Constant Communication ORAM with Small Blocksize," in *Proc. CCS*, 2015.

[19] E. Stefanov and E. Shi, "ObliviStore: high performance oblivious cloud storage," in *Proc. S&P*, 2013.

[20] J. Dautrich and C. Ravishankar, "Combining oram with pir to minimize bandwidth costs," *Proc. CODASPY*, 2015.

[21] B. Chen, H. Lin, and S. Tessaro, "Oblivious parallel ram: Improved efficiency and generic constructions," *IACR Cryptology ePrint Archive*, 2015.

[22] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion oram: A constant bandwidth blowup oblivious ram," *Proc. Theory of Cryptography Conference*, 2015.

[23] J. Dautrich, E. Stefanov, and E. Shi, "Burst oram: Minimizing oram response times for bursty access patterns," *Proc. USENIX Security*, 2014.

[24] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," *Proc. CCS*, 2013.

[25] T. Hoang, C. Ozkaptan, A. Yavuz, J. Guajardo, and T. Nguyen, "S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing," *Proc. CCS*, 2017.