SE-ORAM: A Storage-Efficient Oblivious RAM for Privacy-Preserving Access to Cloud Storage

Qiumao Ma*, Jinsheng Zhang*, Yang Peng$, Wensheng Zhang*, and Daji Qiao*
*Iowa State University, Ames, IA, USA 50011
$University of Washington Bothell, Bothell, WA, USA 98011
Email:{qmma, alexzjs, wzhang, daji}@iastate.edu, yangpeng@uw.edu

***Abstract*—Oblivious RAM (ORAM) is a security-provable approach for protecting clients' access patterns to remote cloud storage. Recently, numerous ORAM constructions have been proposed to improve the communication efficiency of the ORAM model, but little attention has been paid to the storage efficiency. The state-of-the-art ORAM constructions have the storage overhead of $O(N)$ or $O(N \log N)$ blocks at the server, when N data blocks are hosted. To fill the blank, this paper proposes a storage-efficient ORAM (SE-ORAM) construction with configurable security parameter λ and zero storage overhead at the server. Extensive analysis has also been conducted and the results show that, SE-ORAM achieves the configured level of security, introduces zero storage overhead to the storage server (i.e., the storage server only storages N data blocks), and incurs $O(\log N)$ blocks storage overhead at the client, as long as $\lambda \geq 2$ and each node on the storage tree stores $4 \log N$ or more data blocks.**

Key words: Cloud System, Data Outsourcing, Oblivious RAM, Privacy Preservation, Access Pattern.

I. Introduction

Cloud storage services such as Amazon S3 and Dropbox, have been popularly utilized by business and individual clients to host their data. Due to security and privacy concerns, the clients may encrypt their sensitive data before outsourcing them. Nevertheless, data encryption itself is insufficient for data security, because the secrecy of data can still be exposed if a client's access pattern to the data is revealed [1].

The oblivious RAM (ORAM) model [2], which continues shuffling data as the data are accessed, has been a well-known security-provable approach for access pattern protection. Many ORAM constructions [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18] have recently been proposed to make the ORAM model more feasible in practice.

Most of these research have focused on the communication efficiency improvement, but the storage efficiency has not received much attention. To host N data blocks, in general, the state-of-the-art ORAM constructions need the storage server to also store $O(N)$ or $O(N \log N)$ dummy data blocks; in particular, as the most communication-efficient ORAM constructions, Path-ORAM [13] and SCORAM [18] each requires the server to keep $5N$ dummy blocks, and a server running P-PIR [19] has to store $(2 \log N - 1)N$ dummy blocks. Though the unit price for storage is cheaper than that for communication, the significantly-enlarged demand for storage capacity due to the high storage overhead ratio could pose as a high monetary cost to the client, especially when the client needs to have a huge amount of data kept for a long period of time, and when the data need to be replicated in multiple copies for redundancy. Hence, reducing the storage overhead is also imperative.

This study aims to design an ORAM construction with *zero storage overhead* at the server. The research is based on the following observations. The security goal of an ORAM construction is to prevent the storage server from correctly inferring a client's private data access sequence from the client's storage location access sequence that the server can observe. Existing ORAM constructions target at perfect security; that is, the probability is at most $\frac{1}{N^n}$ for the server to correctly infer a sequence with n data accesses from any observed location access sequence, since N^n is the total number of sequences with n data accesses. To attain this goal, the client's query and shuffling operations should be fully random and independent of each other.

Particularly, let us consider the tree-based ORAM [12]. When a data block is assigned to a path of the storage tree, the path is selected uniformly at random to make the query process appear fully random. During an eviction process, nodes are randomly selected to evict data, and each selected node is dictated to evict a data block to its left or right child with the equal probability. Due to the randomness, following *undesired situation* may happen: a node without any real data

978-1-5090-0946-6/16 $31.00 © 2016 IEEE
DOI 10.1109/CSCloud.2016.24

block evictable to its left (or right) child is selected to evict data to left (or right). To deal with such situation, dummy blocks are pre-introduced into the storage when the system is initialized; and it has been shown that, $O(N)$ or $O(N \log N)$ dummy blocks are needed to keep a low failure probability, i.e., the probability that a node has already used up dummy blocks when it is in the afore-described undesired situation.

To address the above issue without introducing storage overhead to the server, we design a new eviction algorithm based on the following intuitions: (i) *Eviction with Non-uniform Probabilities.* When a node is selected to evict data to its children, it can use different probabilities for different children; i.e., a larger probability to evict data to its left child if more of its data blocks are evictable to left, and vice versa. This way, the chance could be significantly reduced for the afore-mentioned undesired situation to occur. (ii) *On-demand Introduction of Dummies.* Nevertheless, the undesired situation could still occur. To deal with it, a dummy block (evictable to both left and right) is inserted *on demand* to replace a real data block, which is moved to the client's cache. Note that, the storage server still stores the same number of data blocks, though some of the blocks become dummies. (iii) *Periodical Removal of Dummies.* As the system keeps running, more dummy blocks are inserted to the server and the client's cache may overflow. To address this issue, an extra query and eviction process is launched periodically to retrieve and discard a dummy from the server and evict a real data block from the client to the server.

Due to the non-uniform eviction probabilities used in the eviction algorithm, perfect security is not attained. To quantify the level of security that our new ORAM construction can achieve, we propose a more generic security definition, which quantify security level by a parameter λ: if an ORAM construction is secure with parameter λ, the probability is at most $(\frac{1}{N^n})^{1-\frac{1}{\lambda}}$ for the server to correctly infer a sequence with n data accesses from any storage location access sequence. That is, the advantage for the server to discover a client's access pattern is upper-bounded by $(\frac{1}{N^n})^{1-\frac{1}{\lambda}} - \frac{1}{N^n}$, which decreases as λ increases. We argue that, this notion of security can be useful in practice, particularly when a large number of data blocks are outsourced and/or protecting relatively long access patterns (i.e., n is large) is the major security goal. For example, when $N = 2^{40}$ and $n = 10$ (or $N = 2^{10}$ and $n = 40$), and $\lambda = 2$, the server's advantage is upper-bounded by 2^{-200}, which may be considered "negligibly small" in practice. Besides, the definition allows a client of our ORAM construction to configure her desired level of security, and manage the tradeoffs between security and performance.

Based on the new eviction algorithm and the new definition of security, we formalize a generic SE-ORAM construction with parameter λ. Through rigorous security and cost analysis, we show that the construction is secure under the definition, and the number of introduced dummy blocks is no more than $x \log N$ with probability $1 - \frac{1}{N^{2x}}$, as long as $\lambda \geq 2$ and each node on the storage tree can store $4 \log N$ or more data blocks. We also instantiate a SE-ORAM construction by setting $\lambda = 2$, analyze its performance, and compare it with the state-of-art ORAM constructions. To summarize, this study makes the following contributions: (i) We introduce a generic security definition for ORAM constructions. It allows a client to configure a desired security level and manage the tradeoffs between security and performance. (ii) We propose SE-ORAM, a generic storage-efficient ORAM construction with configurable security parameter λ. Rigorous analysis shows that, SE-ORAM achieves the configured level of security, introduces zero storage overhead to the storage server (i.e., the storage server only storages N data blocks), and incurs $O(\log N)$ blocks storage overhead at the client, as long as $\lambda \geq 2$ and each node on the storage tree stores $4 \log N$ or more data blocks.

In the rest of the paper, Section II presents the security definition. Section III presents the basic design of SE-ORAM. Section IV compares SE-ORAM with related works. Finally, Section V concludes the paper. Note that, we have also conducted security and cost analysis, which can be found in our technical report [20].

II. Security Definition

Let $\lambda > 1$ be a security parameter. A client exports N equal-size data blocks to a remote storage server. Each data access from the client, which should be kept private, is one of the following two types: (i) read a data block D of unique ID i from the storage, denoted as a 3-tuple $(read, i, D)$; (ii) write a data block D of unique ID i to the storage, denoted as a 3-tuple $(write, i, D)$. To accomplish each data access, the client needs to access some storage location(s) at the remote storage server. Each location access, which can be observed by the server, is one of the following types: (i) retrieve (i.e., read) a data block D from a location l, denoted as a 3-tuple $(read, l, D)$; (ii) upload (i.e., write) a data block D to a location l, denoted as a 3-tuple $(write, l, D)$.

We assume the remote storage server is honest but curious; that is, it stores data and serves the client's location access requests honestly, but it may attempt to figure out the client's data access pattern hidden behind the location accesses. The network connection between the client and the server is assumed to be secure; in practice, this can be achieved using well-known techniques such as SSL [21].

We define the security of our proposed SE-ORAM(λ, N), which has security parameter λ and stores N real data blocks, as follows.

Definition *In SE-ORAM(λ, N), let $\vec{x}_n = \langle\ (op_1,\ i_1,\ D_1),\ (op_2,\ i_2,\ D_2),\ \cdots,\ (op_n,\ i_n,\ D_n)\ \rangle$ denote a private sequence of the client's n data accesses, where each op_i is either a $read$ or $write$ operation; let random variable $A(\vec{x}_n)$ denote the sequence of location accesses (observable by the server) that the client uses to accomplish data access sequence $\vec{x}_n$. Note that, there may exist multiple location access sequences that can accomplish $\vec{x}_n$, each with certain probability to be used by the client as $A(\vec{x}_n)$; hence, $A(\vec{x}_n)$ is a random variable.*

Let $\mathcal{X}_n$ denote the set of all possible sequences of the client's n data accesses, and $\mathcal{A}_n$ the set of all location access sequences that can accomplish at least one data access sequence in $\mathcal{X}_n$.

Let $Pr[\vec{T}_n|\vec{A}_n]$, where $\vec{A}_n \in \mathcal{A}_n$ and $\vec{T}_n \in \mathcal{X}_n$, denote the conditional probability of $A(\vec{T}_n) = \vec{A}_n$ given that $\vec{A}_n$ has been observed by the server.

SE-ORAM(λ, N) is said to be secure if $\forall \vec{A}_n \in \mathcal{A}_n$ and $\forall \vec{T}_n \in \mathcal{X}_n$:

$$(\frac{1}{N^n})^{1+\frac{1}{\lambda}} \leq Pr[\vec{T}_n|\vec{A}_n] \leq (\frac{1}{N^n})^{1-\frac{1}{\lambda}}. \qquad (1)$$

Note that, if the client's data access pattern is perfectly protected, $Pr[\vec{T}_n|\vec{A}_n] = \frac{1}{N^n}$; i.e., no matter what location access sequence (that can accomplish a certain sequence with n data accesses) has been observed, it is impossible for the server to infer the client's actual data access sequence hidden behind this observed pattern, because each of the N^n data access sequences has the same probability $\frac{1}{N^n}$ to be the one. According to the above definition, when $\lambda \to \infty$, $Pr[\vec{T}_n|\vec{A}_n] \to \frac{1}{N^n}$ indeed.

In general, if an SE-ORAM(λ, N) is secure, the advantage for the server to infer the client's actual data access sequence $\vec{T}_n$ from a location access sequence $\vec{A}_n$ that has been observed, i.e., $|Pr[\vec{T}_n|\vec{A}_n] - \frac{1}{N^n}|$, is upper-bounded by $(\frac{1}{N^n})^{1-\frac{1}{\lambda}} - \frac{1}{N^n}$; the larger is λ, the smaller is the bound. Hence, parameter λ quantifies the level of security that an SE-ORAM construction can attain.

III. The SE-ORAM Construction

This section elaborates the SE-ORAM construction in terms of storage organization, data query and data eviction algorithms.

A. Storage Organization and Initialization

1) Server-side Storage: In the server, the storage is initially organized as a complete binary tree. Each node on the tree can store up to s data blocks, where s is a system parameter and an even number. To simplify presentation, we denote the height of tree as h and assume the total number of data blocks N as $N = s \cdot \sum_{l=0}^{h} 2^l = s(2^{h+1} - 1)$. Hence, the number of level-h nodes is 2^h, which also is $\frac{N/s+1}{2} \approx \frac{N}{2s}$.

The content of each data block B_i is encrypted probabilistically with a symmetric cipher (e.g., AES) before the blocks are randomly distributed to the nodes on the tree. Specifically, denoting the plain-text content of a block B_i as D_i, we have $B_i = E(r|D_i)$, where r is a nonce and E is a symmetric encryption function.

In each node n, blocks are randomly divided into two equal-size groups, called *left group* and *right group* and denoted as $G_L(n)$ and $G_R(n)$. Each block in the left group randomly picks a level-h node n' from the left branch of n, and the block is restricted to be evictable toward node n' only; hence, we call the ID of node n' as the *path ID* of the data block. Similarly, each block in the right group also randomly selects a level-h node from the right branch of n, whose ID becomes the block's path ID.

As the data query and eviction processes go on, the tree may become incomplete and some nodes may become non-full (i.e., containing less than s data blocks). Figure 1(a) shows an example of the server-side storage. Here, $h = 3$, two of the level-h nodes (i.e., $n_{3,1}$ and $n_{3,6}$) are absent, and one level-h node (i.e., $n_{3,2}$) is non-full. Also, the data blocks with path IDs of $n_{3,0}$, $n_{3,4}$ and $n_{3,7}$ cannot be completely contained in nodes between level 0 to level 3; hence, *supplementary nodes* have been introduced to provide additional storage, e.g., $n_{4,0}$ for $n_{3,0}$, $n_{4,4}$ and $n_{5,4}$ for $n_{3,4}$, and $n_{4,7}$ for $n_{3,7}$.

2) Client-side Storage: The client-side storage includes three parts: (i) an *index table* $\mathcal{I}$ maintaining the mapping between data block IDs and their path IDs (therefore it has N entries and each entry has h bits); (ii) a *data block cache* $\mathcal{C}$ used to cache data blocks; and (iii) a small *secret storage* storing the key for symmetric data encryption.

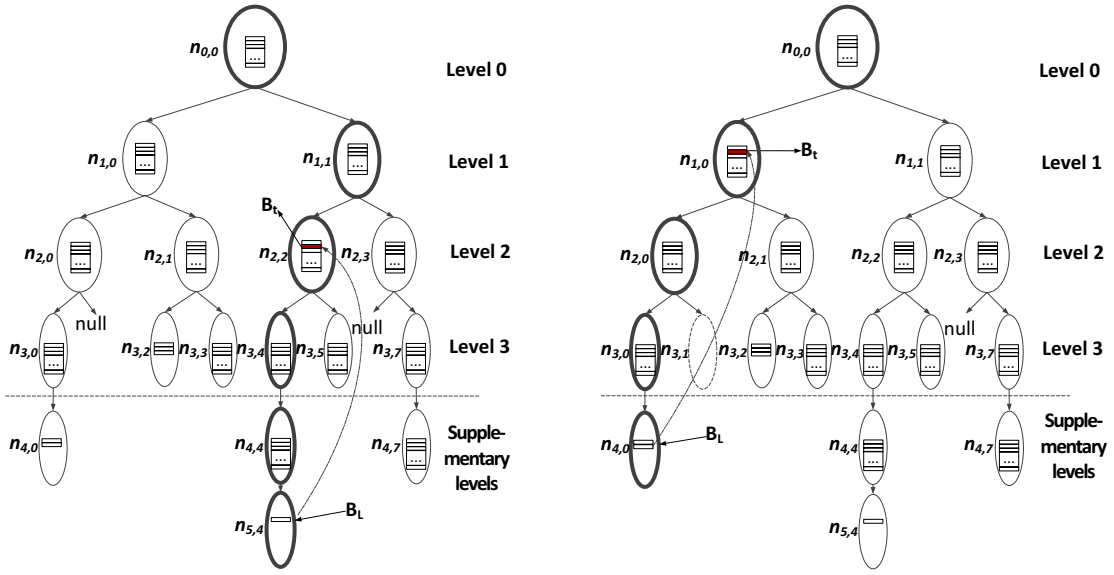

Figure 1. Query Examples. In (a), query target B_t is at node $n_{2,2}$ and has path ID $n_{3,4}$. Node $n_{3,4}$ exists on the tree and has two supplementary nodes. The client requests the server to retrieve nodes from the root to $n_{5,4}$ which is the further supplementary node of $n_{3,4}$. Then, B_L obliviously replaces B_t; finally, as node $n_{5,4}$ becomes empty after B_L has moved, the node is removed from the tree. In (b), query target B_t is at node $n_{1,0}$ and has path ID $n_{3,1}$. Node $n_{3,1}$ does not exist on the tree. The client requests the server to retrieve nodes on the path from the root to $n_{4,0}$, which is the longest path that has the largest overlap with the path from the root to $n_{3,1}$. Then, the client obliviously replaces B_t with B_L.

B. Data Query

When the client queries a data block of ID t (denoted as B_t), it first checks whether B_t is in $\mathcal{C}$; if so, the block is accessed and retained in $\mathcal{C}$. Otherwise, the client looks up the index table $\mathcal{I}$ to obtain B_t's path ID (i.e., the ID of a level-h node, denoted as n_t^h hereafter). Then, the client follows the steps below to obliviously retrieve B_t.

The client requests the server to return data blocks on the path from the root to the n_t^h. In response, the server first finds out all the nodes that should be returned to the client, based on the current topology of the tree: (i) *Case I* - if node n_t^h is currently on the tree and has no supplementary nodes, all the nodes along the path from the root to n_t^h should be returned. (ii) *Case II* - if n_t^h is currently on the tree and has supplementary nodes, all the nodes along the path from the root to n_t^h as well as all of n_t^h's supplementary nodes should be returned. (iii) *Case III* - if node n_t^h is absent, the server acts as follows. Let $n_t^{h_0}$ denote the node that is on the path from the root toward n_t^h (as if n_t^h were still there) and the furthest away from the root. Let $n_t^{h_1}$ denote the leaf node of the longest branch within the subtree rooted at $n_t^{h_0}$. Note that, the path from the root to $n_t^{h_1}$ is the longest path that has the largest overlap with the path from the root to n_t^h (as if n_t^h were still there). All the nodes along the path from the root to $n_t^{h_1}$ should be returned. Let us denote the nodes that should be returned as $n_t^0,\ n_t^1,\ \cdots,\ n_t^L$, where n_t^0 is the root and n_t^L is the leaf node. Among them, suppose node n_t^y on layer y contains B_t.

The server returns only the blocks in n_t^L in the first round. If B_t is among the blocks, the client keeps B_t locally, re-encrypts the rest of the blocks and uploads them back to the server; otherwise, one arbitrary block denoted as B_L is picked from the returned blocks, and the rest of the blocks are re-encrypted and uploaded back to the server.

Next, the server returns all the blocks in n_t^{L-1}. If B_t is among the blocks, the block is kept locally, and the rest of the blocks in n_t^{L-1} together with B_L are re-encrypted and uploaded back to the server. Otherwise, all the blocks in n_t^{L-1} are re-encrypted and uploaded to the server. This process continues until all the blocks on the selected path have been returned to the client, re-encryption and finally uploaded back to the server. Figure 1 shows two examples of data query.

C. Data Eviction

Data eviction should be conducted following the query process, to store the query target B_t back to the server obliviously.

A path (i.e., a level-h node) is selected uniformly at random for B_t, and then all the data blocks on the path are retrieved node-by-node. The eviction process should place B_t into a node on the selected path before the blocks are all re-encrypted and uploaded back to the server. The ID of the path becomes the new path ID of B_t and hence should be recorded in the client's index table $\mathcal{I}$. During the course of eviction, some other blocks may be moved; the movement should ensure that, *a data block stays in a node on the path specified by its path ID or it stays in the local cache maintained by the client*. The eviction steps are elaborated in the following.

a) E1 - Initial Step: Let B_e denote the current block to evict (called the evicted block), and n_e the current node (called the evicting node) to accommodate B_e's eviction. Initially, $B_e = B_t$ and $n_e =$ root. All the data blocks in n_e are sent from the server to the client.

b) E2 - Conditional Termination: If n_e is non-full, the client writes B_e into n_e. Then, B_e is put into the left or right group of n_e (i.e., $G_L(n_e)$ or $G_R(n_e)$) according to its path ID; note that, if B_e is a dummy, it is randomly put into either $G_L(n_e)$ or $G_R(n_e)$.

Another condition for the process to terminate is when n_e is a level-h node. B_e should be written to the furthest supplementary node of n_e. If the supplementary node is full, an additional supplementary node is created to contain B_e.

For both cases, blocks in n_e (and its supplementary nodes if applicable) should be re-encrypted and uploaded back to the server.

c) E3 - Selection of the Next Evicting Node: Depending on the sizes of $G_L(n_e)$ and $G_R(n_e)$, the selection of the next evicting node (denoted as n'_e) works as follows:

If $|G_L(n_e)| > |G_R(n_e)|$, the left child of n_e is selected as n'_e with probability $1-p$ while the right child is selected as n'_e with probability p, where $p = \frac{1}{2^{1/\lambda}+1}$ and $1-p = \frac{2^{1/\lambda}}{2^{1/\lambda}+1}$.

If $|G_L(n_e)| = |G_R(n_e)|$, the left and right children of n_e have the same probability 0.5 to be selected as n'_e.

If $|G_L(n_e)| < |G_R(n_e)|$, the left child of n_e is selected to be n'_e with probability p while the right child is selected to be n'_e with probability $1-p$.

Note that, if n'_e does not exist on the tree, it should be created: 1) If n_e is a level-h node or a supplementary node, a supplementary node n'_e is created and linked to n_e; 2) otherwise, n'_e is created as a left or right child node of n_e accordingly.

d) E4 - Selection of the Next Evicted Block: There are a few different cases. Case I - If B_e is a dummy block, it remains to be the next evicted block denoted as B'_e. Case II - If B_e is a real data block, and there is at least one block in $n_e \cup \{B_e\}$ (we also use n_e to denote the set of all data blocks in n_e, for simplicity) that is evictable to n'_e, one such block is selected to be B'_e and the selected block is replaced by B_e. Case III - If B_e is a real data block, no data blocks in $n_e \cup \{B_e\}$ are evictable to n'_e, but n_e contains dummy blocks, one dummy block is selected as B'_e and the selected block is replaced by B_e. Case IV - If B_e is a real data block, no data blocks in $n_e \cup \{B_e\}$ are evictable to n'_e, and n_e does not contain any dummy blocks, a new dummy block is created to be B'_e, while the original B_e is saved to the client's local cache. Finally, all current blocks in n_e are re-encrypted and uploaded back to the server; then, after $B_e \leftarrow B'_e$ and $n_e \leftarrow n'_e$ are performed, the process continues to Step E2.

D. Extra Query-Eviction Round

With the above eviction algorithm, the dummy blocks at the storage server and the cached blocks at the client may keep increasing as more data blocks are queried. To bound the number of these blocks and hence the storage overhead, we propose to periodically remove dummy blocks and dump cached data blocks as follows.

Every time after an eviction process is completed, with probability ρ, the following extra round of query and eviction is conducted: The client randomly selects a path. Depending on the selected path, this step proceeds with one of the following two cases. Case I - the selected path contains dummy blocks. In this case, one dummy block is retrieved from the selected path following the above data query algorithm. Then, one real data block is randomly picked from the client's cache, and evicted to the tree structure at the storage server following the above data eviction algorithm. Case II - the selected path does not contain any dummy blocks. In this case, one data block is randomly retrieved from the selected path following the above data query algorithm, and then evicted following the above data eviction algorithm.

IV. Performance Comparison

We instantiate the generic SE-ORAM by setting $\lambda = 2$, $c = 1$ and thus $s = 4 \log N$, and compare the instantiated SE-ORAM with several state-of-the-art ORAMs including T-ORAM [12], G-ORAM [16], Path ORAM [13], SCORAM [18], and P-PIR [19], in terms of storage and communication overheads.

Table I compares SE-ORAM with state-of-the-art ORAM constructions in terms of the client and server

Table I
STORAGE AND COMMUNICATION OVERHEADS. N: NUMBER OF DATA BLOCKS; B: BLOCK SIZE IN BITS. SERVER STORAGE OVERHEAD IS DEFINED AS THE SERVER'S STORAGE CONSUMPTION OTHER THAN THE NB BITS FOR THE REAL DATA BLOCKS. CLIENT STORAGE OVERHEAD IS DEFINED AS THE CLIENT'S STORAGE CONSUMPTION OTHER THAN THE INDEX TABLE FOR THE EXPORTED DATA BLOCKS.

ORAM	Client Storage Overhead	Server Storage Overhead	Communication Overhead
T-ORAM [12]	$O(B)$	$O(N \log N \cdot B)$	$(\log^2 N \cdot B)$
G-ORAM [16]	$O(\log^2 N \cdot B)$	$O(N \cdot B)$	$(\frac{\log^2 N}{\log \log N} \cdot B)$
Path ORAM [13], SCORAM [18]	$O(\log N \cdot B)$	$O(N \cdot B)$	$O(\log N \cdot B)$
P-PIR [19]	$O(B)$	$O(N \log N \cdot B)$	$O(\log N \cdot B)$
SE-ORAM	$O(\log N \cdot B)$	0	$O(\log^2 N \cdot B)$

storage overheads as well as the communication overhead per query. As we can see, SE-ORAM does not consume any extra storage in the server other than $N \cdot B$ bits for the N data blocks. On the contrary, the server storage overhead of each of the state-of-the-art ORAM constructions is $O(N \cdot B)$ or $(N \log N \cdot B)$ bits. Though the communication cost of SE-ORAM is on the same level as T-ORAM, it can be reduced to $O(\log N \cdot B)$ by adopting the additive Homomorphic encryption-based PIR primitives [19], similar to the way that P-PIR reduced the communication cost of T-ORAM from $O(\log^2 N \cdot B)$ to $O(\log N \cdot B)$.

V. CONCLUSION

In this paper, we introduce a generic security definition for ORAM constructions, which allows a client to configure a desired security level and manage the tradeoffs between security and performance. We also propose SE-ORAM, a generic storage-efficient ORAM construction with configurable security parameter λ. SE-ORAM achieves the configured level of security, introduces zero storage overhead to the storage server (i.e., the storage server only storages N data blocks), and incurs $O(\log N)$ blocks storage overhead at the client, as long as $\lambda \geq 2$ and each node on the storage tree stores $4 \log N$ or more data blocks.

ACKNOWLEDGEMENT

This work is partly sponsored by NSF under Grant CNS-1422402.

REFERENCES

[1] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation," in *Proc. NDSS*, 2012.
[2] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, May 1996.
[3] M. T. Goodrich and M. Mitzenmacher, "Mapreduce parallel cuckoo hashing and oblivious RAM simulations," in *Proc. CoRR*, 2010.
[4] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *Proc. SODA*, 2012.
[5] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *Proc. ICALP*, 2011.
[6] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious RAM simulation with efficient worst-case access overhead," in *Proc. CCSW*, 2011.
[7] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *Proc. SODA*, 2012.
[8] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Proc. CRYPTO*, 2010.
[9] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *Proc. CCS*, 2008.
[10] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: a parallel oblivious file system," in *Proc. CCS*, 2012.
[11] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proc. CCS*, 2012.
[12] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *Proc. ASIACRYPT*, 2011.
[13] E. Stefanov, M. V. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *Proc. CCS*, 2013.
[14] E. Stefanov and E. Shi, "ObliviStore: high performance oblivious cloud storage," in *Proc. S&P*, 2013.
[15] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *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," in *Proc. PETS*, 2013.
[17] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *Proc. CCS*, 2013.
[18] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "SCORAM: oblivious RAM for secure computation," in *Proc. CCS*, 2014.
[19] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient private file retrieval by combining ORAM and PIR," in *Proc. NDSS*, 2014.
[20] Q. Ma, J. Zhang, W. Zhang, and D. Qiao, "SE-ORAM: A Storage-Efficient Oblivious RAM for Privacy-Preserving Access to Cloud Storage," *IACR Cryptology ePrint Archive*, vol. 2016, no. 256, 2016.
[21] A. Freier, P. Karlton, and P. Kocher, "The secure sockets layer (SSL) protocol version 3.0," in *RFC 6101*, 2011.