IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 13, 2019, accepted June 23, 2019, date of publication June 28, 2019, date of current version July 15, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2925789

Radix Path: A Reduced Bucket Size ORAM for Secure Cloud Storage

KHOLOUD SAAD AL-SALEH, (Member, IEEE), AND ABDELFETTAH BELGHITH, (Senior Member, IEEE)

College of Computer and Information Science, King Saud University, Riyadh 12372, Saudi Arabia

Corresponding author: Abdelfettah Belghith (abelghith@ksa.edu.sa)

This work was supported by the King Abdulaziz City for Science and Technology (KACST) under Grant 1-17-00-001-0004.

ABSTRACT This paper proposes a novel version of path oblivious random access memory called radix path ORAM (R-Path ORAM) with a large root (radix) bucket size but a small fixed size for all the other buckets in the tree. A detailed analysis of the root bucket occupancy is conducted to provide a closed-form solution of the required root bucket size that maintains a negligible failure probability. The performance of the R-Path ORAM is evaluated and compared against the traditional Path ORAM using a unified platform. The conducted experiments clearly show that R-Path ORAM provides much lower server storage and average response time than the seminal Path ORAM. Furthermore, we propose a background eviction technique to eventually reduce the root bucket size and avoid system failure. The conducted experiments on the unified platform showed the usefulness and efficiency of the proposed two-way eviction technique in successfully reducing the root bucket size while incurring a very small overhead.

INDEX TERMS Cloud storage, ORAM, path ORAM, security, system failure.

I. INTRODUCTION

Cloud storage is being heavily adopted nowadays. However, storing sensitive data in the cloud has many security risks [1], [2]. There are many technologies tailored to solve and address the security risks associated with cloud storage. One of the technologies that stands out is searchable encryption. In searchable encryption both data and queries are encrypted. The cloud server upon receipt of an encrypted query performs the search on the encrypted data and returns the encrypted result to the user. As such, full confidentiality is enforced as everything is encrypted [3]. However, observing the access pattern to the stored encrypted data may reveal nontrivial information about the content of that data. Indeed given some background knowledge about the stored encrypted data, the authors in [4] showed that as much as 80% of the data can be inferred by observing and analyzing the access pattern.

Fortunately, there is a cryptographic primitive that can be used to implement searchable encryption and at the same time hide the access pattern. This cryptographic primitive is Oblivious RAM (ORAM) [5]. ORAM was first proposed by Ostrovsky [6] and Ostrovsky and Goldreich [7], since then there have been many developments in the field of ORAMs [8]–[16].

In general, ORAMs can be divided into two categories based on how the data is stored on the server. The first category is the hierarchical ORAM, that uses a hierarchical data structure to store the data on the server and the second category is tree ORAMs that uses a tree data structure. Tree ORAMs are the most recent and generally outperform hierarchical ORAMs [17], [18].

There are a number of tree ORAMs among them are Path, Ring, XOR Ring and Onion. However, Path ORAM outperforms the other three ORAMs in terms of performance and simplicity as shown by AlSaleh and Belghith [19]. The original Path ORAM uses a binary tree structure to store the data at the server and the nodes of the binary tree are called buckets. A bucket can hold a constant number, say z, of data blocks that should be greater or equal to 4 to ensure that Path ORAM has negligible failure probability. Stevanov *et al.* [14] proved this theoretically for $z = 5$ and experimentally for $z = 4$. Furthermore, in Path ORAM to access a given data block, a complete path of the tree starting from the root bucket until the leaf bucket should be read and then written back. During such an access, all the z data blocks of all the buckets on the path are read and then written back.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Asif.

This paper proposes a new version of Path ORAM that uses a smaller bucket size, z, to reduce the amount of storage needed at the server side as well as to reduce the number of blocks in an accessed path. The reduction of the number of blocks in an accessed path tacitly leads to a valuable improvement in the average response time. However, the reduction of the bucket size affects the failure probability that should be maintained at a negligible value. To this end, we analyze the situation that leads to Path ORAM failures, namely the situation where an overflow occurs at the root bucket. The root bucket size should be much larger than z. We provide an accurate estimation of the needed maximum root bucket size. Then using this information, we propose a new version of Path ORAM coined Radix Path ORAM that uses a large size for the root bucket and a smaller bucket size z for the rest of the buckets in the tree. We compare the performance of Radix Path ORAM against the performance of the original Path ORAM using a unified experimental platform. The conducted experimental results show that the average response time and the server storage are significantly reduced. Moreover, we propose a background eviction technique to enable our new enhanced version of Path ORAM to use a lower root bucket size and at the same time avoid the possibility of an ORAM failure. The contributions of this paper are:

1) Mathematical modeling and analysis of the root bucket occupancy in Path ORAM and its experimental validation.
2) Introduction of Radix Path (R-Path) ORAM: a novel improvement to Path ORAM with a large root bucket size and its performance evaluation.
3) Development of an efficient background eviction technique to enable the reduction of the root bucket size.
4) Integration of the background eviction technique into the original Path ORAM and the R-Path ORAM, and the experimental assessment of its practical impact.

The paper is organized as follows: in section II, we provide a detailed description of the original Path ORAM. Then in section III, we present some of the related work. In section IV, we specify the details of the unified experimental platform used to assess and compare the performance of the proposed R-Path ORAM to that of the original Path ORAM. In section V, we provide a detailed analysis of the root bucket occupancy needed to sustain the lowering of the bucket size z. In section VI, we propose a new version of Path ORAM, the R-Path ORAM that uses a large root bucket size while fixing a smaller bucket size for the rest of the buckets. Section VII introduces a simple yet efficient eviction technique that can be used to reduce the root bucket size as well as to completely avoid failure situations. Finally, section VIII concludes the paper.

II. PATH ORAM

Path ORAM uses a binary tree data structure to store the data at the server. Each node in the binary tree is called a bucket and can hold up to a constant number of data blocks referred to by the constant z. In Path ORAM each data block is mapped to a leaf of the binary tree and this mapping is saved in a data structure stored at the client side called the leaf map or position map. When a data block is mapped to a leaf bucket, this means that the data block can only be stored in one of the buckets on the path from the root bucket until the leaf bucket.

The client in Path ORAM has a local buffer called a stash to store the read data blocks. The main invariant in Path ORAM is that at any time, a data block is either in the tree on the path it has been mapped to, or in the stash waiting to be mapped to one of the buckets of its designated path.

To access a data block, the position map is firstly consulted to find the leaf to which the data block is mapped, say x, then the complete path starting from the root to leaf x is completely read from the binary tree and stored in the stash. The accessed data block is then presented to the user for reading or updating. This accessed data block is then re-mapped to a new random leaf and the position map is updated accordingly. At this point, the read path is ready to be written back from the stash into the binary tree at the server according to the greedy filling strategy.

The greedy filling strategy starts from the leaf bucket and goes upward in the path level by level until the root bucket. Normally all the read data blocks from the path except the accessed block (i.e. target block) can be written back into the path. To be able to write a data block in a bucket, the bucket must have an empty slot and must be on the path from the root to the leaf node to which this data block is mapped. Each bucket contains exactly z slots, where each slot is either a data block or a dummy block. The data blocks written back in the path, may include the accessed data block as well as any additional blocks already in the stash from previous read paths, as long as the Path ORAM invariant is satisfied [13], [20]. Of course, when a data block is written back into the tree it is removed from the stash.

The binary tree is of height L with the root being at level 0 and the leaves being at level L. Hence the tree has 2^L leaves numbered from 0 to $2^L - 1$. Following Stefanov *et al.* [14], [20], we consider that the total number of nodes (i.e., buckets) of the tree is denoted by N, and as such $L = log(N + 1) - 1$. The symbols used for Path ORAM are shown in Table 1 and the Path data access algorithm is shown in Algorithm 1.

Path ORAM has a negligible failure probability only if the bucket size $z \geq 4$. Stevanov *et al.* [14] proved this theoretically for $z = 5$ and experimentally for $z = 4$. The reason for this is when $z < 4$, the number of left over blocks in the stash keeps increasing which results in a total number of stored blocks exceeding the size of the stash, and consequently the ORAM system fails. Ren *et al.* [21] argued that Path ORAM with $z \leq 2$ consistently fails, despite using a very big stash of 1000 blocks. They also showed that Path ORAM with the same large stash size, will fail with non-negligible probability when $z = 3$.

To provide a better understanding as to how blocks keep accumulating in the stash, we divide it into two parts. A temporary part, called stash1, that holds the read path and a

TABLE 1. Symbols used for Path ORAM.

N	Number of data blocks
L	Tree height
z	Number of blocks in a bucket
oper	Operation R for Read and W for Write
add	Address of data block to be read or written
data*	Holds the data if the operation is a write and is empty if the operation is a read
P(x)	Path from the root to the leaf node x
P(x, i)	Bucket on path x at level i
S	Client's stash
PosMap	Position map saved on the client
x:=PosMap[add]	Block with address add is mapped to leaf node x
ReBuck(i)	Read a complete bucket i
WrBuck(i, y)	Write bucket i with the contents from y
UniRand	Uniform Random distribution

Algorithm 1 Path ORAM Data Access Algorithm

Input: oper, add, data*

$x \leftarrow PosMap[add]$
$PosMap[add] \leftarrow UniRand(0 \dots 2^L - 1)$
for $i = 0$ to L **do**
  $S \leftarrow (S \cup ReBuck(P(x, i))$
end for
if $(oper = W)$ **then**
  $S \leftarrow (S - (add, data)) \cup (add, data^*)$
end if
for $i = L$ downto 0 **do**
  $S' \leftarrow (add', data') \in S :$
    $P(x, i) = P(PosMap[add'], i)$
  $S' \leftarrow Min(|S'|, z)$ blocks from S'
  $S \leftarrow S - S'$
  $WrBuck(P(x, i), S')$
end for
return data

residual part, called stash2, that holds the left-over blocks after writing the path back to the ORAM (i.e. the blocks that cannot be written back to the ORAM). For Path ORAM with $z = 4$, the residual stash has to be able to accommodate 89 blocks to achieve negligible failure probability [14].

Recall that for any block access request in Path ORAM, a complete path has to be read from the tree, stored in stash1 and then the path has to be written back again using a greedy filling strategy with blocks from stash1 and stash2 that satisfy the Path ORAM invariant. During this process all the blocks of the read path but the requested one will be written back. The accessed block may no longer fit in the path since it has been re-mapped to a new random leaf. Indeed for $z \geq 4$, most of the time the newly accessed block is indeed written back. However, for $z < 4$ it becomes harder to find a bucket that can hold the accessed block, and still satisfies the Path ORAM invariant, and consequently the block has to wait in stash2. This block will have then to wait for the following path writes to be written back into the tree. The blocks waiting in stash2 may wait for several path writing instances resulting in an increase of the size of stash2 (i.e. blocks get accumulated in stash2 waiting to be written back). When the number of blocks in stash2 reaches its maximum and a new access request is performed and the newly accessed block can no longer be written back into the path, Path ORAM then fails [14], [20].

We would like to design a new version of Path ORAM with a smaller bucket size z to reduce the amount of storage needed at the server side as well as to reduce the number of blocks in an accessed path which in turn yields a much lower communication cost and a reduced response time. However, very special attention should be given to the failure probability to remain negligible. Path ORAM fails when stash2 is already full and a newly accessed block cannot be written back into the path. Therefore, we would like to investigate when the root bucket gets full resulting in a block having to wait in stash2. We shall also investigate the maximum size needed for stash2 to have a very negligible probability of failure. Then we shall consider stash2 as residing at the root of the tree. As such, the root will be assigned a large capacity while we may lower the value of z for the rest of the buckets.

III. RELATED WORK

Most of the relevant work on improving the operation and performance of Path ORAM have been done in the context of secure processors. In this paper, we are rather focusing on the cloud storage. In the context of secure processors, the sequence of requests may be considered known in advance which is not the case for cloud storage.

In the context of secure processors, Fletcher *et al.* [22], Fletcher [23], and Ren *et al.* [21] introduced background eviction to lower the probability of Path ORAM failure. However, we will use background eviction as a leverage for R-Path ORAM to eventually reduce the size of the root bucket.

Ren *et al.* [21] also tried to enhance the performance of Path ORAM in the secure processor domain by introducing the static super block technique. Blocks that have some form of locality are gathered together and saved in a super block. Blocks in a super block are all mapped to the same leaf in the Path ORAM tree. Hence, all the blocks that belong to a super block are read whenever a single block in the super block is read. Moreover, after reading the blocks in a super block, not only the single accessed block gets re-mapped but also all the blocks belonging to the same super block get re-mapped to the same random leaf. However, they only used address space locality and not data locality limiting their work to program locality. Moreover, the grouping of the blocks into super blocks is to be done before loading the ORAM tree.

Yu *et al.* [24] proposed using a dynamic super block. Their work is an improvement to the work of Ren *et al.* [21]. The dynamic super block technique permits the contents and size of the super block to change during the run of the ORAM taking into consideration the programs locality. However, the dynamic super block still only takes consideration of program locality and not data locality and is then restricted to

the secure processor domain. They called their new approach Dynamic Prefetcher for ORAM (PrORAM).

Another enhancement to Path ORAM was proposed by Zhang *et al.* [25]. They proposed to merge two successive access requests together since they might share some common buckets in their paths. Unfortunately this necessitates the prior knowledge of the sequence of requests, yet the merging of two requests together resulted in the saving of just one bucket (i.e. only one bucket does not have to be re-read) as showed by Sánchez-Artigas [16]. This saving of one bucket is actually in the root and we could have attained this saving by using root top caching.

Sánchez-Artigas [16] tried to improve the work done by Zhang *et al.* [25] by merging the access requests in larger groups than two. Moreover, Sanchez suggested dynamic reordering of the batch access requests to attain the maximum common buckets amid paths of a batch. The dynamic reordering obviously necessitates the prior knowledge of the sequence of requests which is not feasible for outsourced data.

Al-Saleh and Belghith [26] proposed an enhanced version of Path ORAM that they called Locality Aware Path ORAM. Their new version is used for outsourced data and takes advantage of any eventual locality and popularity in the data itself. They only had to introduce a small extra storage at the client side, and they showed that their new version of Path ORAM outperforms the traditional Path ORAM even with data that has a small popularity rate and using a small cache.

Mass *et al.* [27] were the first to introduce tree top caching. In tree top caching the top k levels of the ORAM tree are cached in the stash. The rationale behind this is to reduce the length of the accessed path, so instead of reading and writing a complete path of length $L + 1$ we only need to read and write a path of length $L - k + 1$ where the top k levels are stored locally at the client. This can improve the performance of the ORAM. They implemented tree top caching on a secure processor named Phantom that uses Path ORAM.

Ren *et al.* [11] also suggested using tree top caching with Ring ORAM. They proposed caching the top k levels of the tree. We will adopt the same technique proposed by Mass *et al.* [27]. However, we will only cache the root bucket as we intend to make it larger than all the other buckets. We name this specific case, root caching.

Our proposed Radix Path ORAM (R-Path) uses a constant bucket size z smaller than 4 for all the buckets but the root. The root bucket size will be fixed to a value that maintains a negligible failure probability. This shall be investigated analytically and experimentally using a developed platform. Furthermore, R-Path ORAM uses root caching at the client side and an efficient background eviction technique based on dummy requests to reduce the root bucket size and to nullify the probability of failure.

IV. UNIFIED EXPERIMENTAL PLATFORM

We set up a unified experimental platform to conduct various experiments on the seminal Path ORAM and our proposed R-Path ORAM, and compare their performance. The platform is also used to conduct experiments to validate our developed mathematical model ascertaining the maximum root size, and to evaluate the performance of the proposed background eviction scheme.

The experimental platform consists of a client, a server and a communication switch connecting the server and client with a base speed of 1.09 Gbps. The client is a laptop with 1 TB hard disk, 8 GB of memory and has Intel core i7-7 500u, up to 3.5 GHz. The server has 1 TB hard disk, 128 GB of memory and is composed of 8 cores. The operating system used in both client and server is Ubuntu 14.04. MongoDB is used at the server to hold and manage the database.

We used the C++ programming language and we implemented the non-recursive version of Path ORAM. The AES/CFB is used for the necessary encryption with a key length of 128 bits. The block size is set to 4096 Bytes. Moreover, we set the number z of blocks in a bucket to 4 for Path ORAM. Requests for Data blocks are assumed to follow the Poisson distribution with an inter arrival time of one minute. Data blocks are chosen randomly. Six hundred random access requests are performed in each experiment. Experiments are repeated ten times to calculate the 95% Confidence Interval (CI).

V. ROOT BUCKET OCCUPANCY

The occupancy (the number of data blocks) of the root emanates from the eventual overflow of its children in addition to data blocks that are successively accessed from the same half of the ORAM tree and re-mapped to paths residing on the other half of the tree. Our objective is to get a closed form solution for the maximum of the root occupancy. The number of blocks that accumulate in the root bucket from data blocks that are successively accessed from the same half of the ORAM tree and re-mapped to paths residing on the other half of the tree will be here after referred to by ''direct overflow''. On the other hand, the number of blocks that overflow from the children into the root bucket will be here after referred to by ''indirect overflow''.

A. DIRECT OVERFLOW

Let X be a random variable counting the number of successive data blocks that should be placed at the root at any time due to direct overflow. We consider the following 4 types of requests or events:

1) Event RR: this represents a request for a data block mapped to a leaf residing on the right half of the tree (as indicated by the Position Map) that is to be re-mapped to a leaf on the right half of the tree.
2) Event LL: this represents a request for a data block mapped to a leaf residing on the left half of the tree (as indicated by the Position Map) that is to be re-mapped to a leaf on the left half of the tree.
3) Event RL: this represents a request for a data block mapped to a leaf residing on the right half of the tree

(as indicated by the Position Map) that is to be re-mapped to a leaf on the left half of the tree.

4) Event LR: this represents a request for a data block mapped to a leaf residing on the left half of the tree (as indicated by the Position Map) that is to be re-mapped to a leaf on the right half of the tree.

The probability of any of these defined events is $\frac{1}{4}$ independently from the tree size and the bucket size (i.e. independent from N and z). $X = k$ means that we have k successive accesses of data blocks mapped to leaves residing on the same half of the tree and randomly re-mapped to leaves of the other half of the tree; that is a sequence of k RL events or a sequence of k LR events. It is very useful here to note that a sequence of RL (respectively LR) events could be interspersed by any number of RR events (respectively LL events). Such an interspersion does not affect the occupancy of the root being accumulated by the sequence of the RL or LR events. On the other hand, the accumulation of the data blocks at the root by a sequence of RL events (respectively LR events) is stopped by the first encountered LL or LR event (respectively RR or RL event) which necessitates the execution of the greedy refilling and rewriting of the accessed path. Consequently, let us consider the random variable Y that counts the successive occurrences of RL (respectively LR) interspersed by occurrences of RR (respectively LL). Consider the following sequence of events (data block accesses) of types RR, LL, RL and LR:

LL LL LR LR LL LL LL LR LL LR RR RR RR RL RL RL RR RR RL RL RR RR RR LR LR LL LR LR LL LL LL LR LR LR LR LL LL RR RR

The first occurrence (blue) of variable Y starts with LR and ends when we first encounter RR. Here Y is equal to 8 events or successive data block accesses, X is equal to 4 successive LR events which amounts to 4 data blocks to be stored in the root. The second occurrence (green) of Y starts with RL and ends when we encounter LR. This amounts to $Y = 10$ and $X = 5$. The third occurrence (red) of Y starts with LR, ends when we encounter RR and amounts to $Y = 14$ and $X = 8$. Let us first concentrate on random variable Y and later on we get back to random variable X. We clearly observe that the occurrence of Y either:

1) starts with LR and ends either with RR or RL, or
2) starts with RL and ends either with LL or LR.

Random variable Y is clearly geometrically distributed:

$$P[Y = k + 1] = q^k p \quad k \geq 0 \tag{1}$$

where $q = \frac{1}{2}$ and denotes the probability for the next event to be RR or RL (respectively LL or LR) given that Y has started with RL (respectively LR) and $p = \frac{1}{2}$ denotes the probability for the next event to be LL or LR (respectively RR or RL) given that Y has started with RL (respectively LR). That is $q = 1 - p$. Notice that we put $P[Y = k+1]$ and not $P[Y = k]$ since the event that started the occurrence is already given.

Now if we consider a very long sequence of data block accesses, we shall have a very big number, denoted by n, of occurrences of the random variable Y. These occurrences are independently and identically (geometrically) distributed (iid). We would like to get a closed form of the expected value of the maximum of these n iid geometric random variables.

1) CLOSED FORM OF THE EXPECTED VALUE OF THE MAXIMUM DIRECT OVERFLOW

First notice that formula (1) can be rewritten as:

$$P[Y = k] = q^{k-1} p \quad k \geq 1 \tag{2}$$

Let $Y_1, Y_2, \ldots, Y_n$ denote our n iid geometric random variables. By formula (2), we have:

$$P[Y_i \leq k] = 1 - q^k \quad i = 1, \ldots, n \; k \geq 1 \tag{3}$$

Now let M denote the random variable accounting for the maximum of the n iid geometric random variables. That is $M = max(Y_1, \ldots, Y_n)$, then using (3) we have:

$$P[M \leq k] = (1 - q^k)^n \quad k \geq 0$$

The expected value of the maximum of the n iid geometric variables is then given by:

$$\bar{M} = \sum_{k=0}^{\infty} P[M > k] = \sum_{k=0}^{\infty} (1 - (1 - q^k)^n)$$

Unfortunately, there is no closed form expression for this sum [28]. However, we might approximate it by its corresponding integral with $q = e^{-\lambda}$, that is:

$$\int_0^{\infty} (1 - (1 - e^{-\lambda x})^n) dx < \sum_{k=0}^{\infty} (1 - (1 - q^k)^n) < 1 + \int_0^{\infty} (1 - (1 - e^{-\lambda x})^n) dx$$

and since [28],

$$\int_0^{\infty} (1 - (1 - e^{-\lambda x})^n) dx = \int_0^1 \frac{1 - x^n}{\lambda (1 - x)} dx = \int_0^1 \sum_{k=0}^{n-1} \frac{x^k}{\lambda} dx = \sum_{k=1}^{n} \frac{1}{\lambda k}$$

It follows that:

$$\frac{1}{\lambda} \sum_{k=1}^{n} \frac{1}{k} < \bar{M} < 1 + \frac{1}{\lambda} \sum_{k=1}^{n} \frac{1}{k}$$

and finally putting back $\lambda = -ln(q)$ gives:

$$\frac{1}{-ln(q)} \sum_{k=1}^{n} \frac{1}{k} < \bar{M} < 1 + \frac{1}{-ln(q)} \sum_{k=1}^{n} \frac{1}{k} \tag{4}$$

We may then confidently approximate the expected value of the maximum of our n iid geometric random variables by:

$$\bar{M} \approx \frac{ln(q) - \sum_{k=1}^{n} \frac{1}{k}}{ln(q)} \tag{5}$$

and for very large n, we may approximate the harmonic number $H_n = \sum_{k=1}^{n} \frac{1}{k}$ using expression 0.131 in [29]; that is:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \gamma + ln(n) + \frac{1}{2\,n}$$

where γ is the Euler's constant and is given by $\gamma = 0.5772156649$. As n is assumed to be very large, we consider the following approximation instead:

$$H_n \approx \gamma + ln(n)$$

and consequently (5) yields:

$$\bar{M} \approx \frac{ln(q) - \gamma - ln(n)}{ln(q)} \tag{6}$$

In a similar way, we approximate the variance of the maximum of our n iid geometric random variables by:

$$Var(M) \approx \frac{1}{ln^2(q)} \sum_{i=1}^{n} \frac{1}{i^2} \tag{7}$$

The harmonic of degree 2, $H_n(2) = \sum_{i=1}^{n} \frac{1}{i^2}$ can be readily approximated for very large values of n using equation 9 of [26] where we put $\alpha = 2$:

$$H_n(2) \approx \frac{6n-1}{4n} \approx \frac{3}{2}$$

Now we are ready to get back to random variable X. The question essentially amounts to ascertain the value of X given that the value of Y is known; that is how many occurrences of RL (respectively LR) are in a given occurrence of Y. It is not difficult to see that X given Y follows the binomial distribution with parameter l (the value of Y) and $p = \frac{1}{2}$; that is:

$$P[X = k/Y = l] = \binom{l}{k} p^k (1-p)^{l-k} \tag{8}$$

The expected value of X, that is the average number of RL (respectively LR) in a given occurrence of Y of length l, is $lp = \frac{l}{2}$. Consequently, the expected value of the maximum of n occurrences of random variable X is given using (6) by:

$$\bar{M_X} \approx \frac{ln(q) - \gamma - ln(n)}{2 \quad ln(q)} \tag{9}$$

While the random variable X is independent from the size of the tree (X does not depend on N the number of nodes in the tree) as well as the size z of the each bucket, the number of data blocks that overflow from the children into the root (i.e. the indirect overflow) does greatly depend on both N and z.

We would like to make a final note, about this mathematical model. We stated previously that the occurrence of Y either:

1) starts with LR and ends either with RR or RL, or
2) starts with RL and ends either with LL or LR.

However, the filling of the root may continue after encountering RR or RL (respectively LL or LR) with a small though a non-null probability. This may happen when the accessed path cannot accommodate any of the blocks in the root bucket.

TABLE 2. Direct overflow for different size z and for different number of requests.

#	100,000		1,000,000		10,000,000	
z	**Av**	**CI**	**Av**	**CI**	**Av**	**CI**
1	10.1	0.54	12.6	0.67	14.2	0.59
2	10.1	0.54	12.6	0.67	14.2	0.59
3	10.1	0.54	12.6	0.67	14.2	0.59
4	10.1	0.54	12.6	0.67	14.2	0.59
5	10.1	0.54	12.6	0.67	14.2	0.59

As such, our proposed mathematical model is rather optimistic and may sometimes yield a lower root occupancy than that resulting from any actual direct overflow.

2) VALIDATION OF THE MATHEMATICAL MODEL

To validate our mathematical model, we performed experiments on our unified platform. We kept track of the maximum number of accumulated blocks in the root bucket resulting from the direct overflow. We needed to eliminate the indirect overflow from the children. Thus, we set up our ORAM to have infinite bucket size for the first top two levels (i.e. the root and its two children) Thus, allowing the root bucket to accommodate as much blocks as necessary without overflowing and at the same time the children buckets have enough space so that they do not overflow buckets to the root. The bucket size for the rest of the levels is set to z like in traditional Path ORAM.

We performed experiments for different bucket size z ranging from 1 to 5 on a tree with L = 10 and N = 1024. Furthermore, we performed the experiments for different numbers of access requests 1,00,000, 1,000,000 and 10,000,000 to purposely show that the number of access requests affects the number of blocks in the root bucket. Each experiment is repeated ten times and we calculated the corresponding confidence interval at 95 %. The obtained results are displayed in Table 2 where AV represents the average maximum direct overflow and CI the obtained confidence interval.

We clearly observe from Table 2 that the direct overflow does not depend on the used bucket size z. The direct overflow, however, is affected by the number of access requests (i.e., the length of the request sequence) that are performed on the ORAM tree; it increases with the increase of the number of access requests.

Further conducted experiments, using larger trees have showed, as we know, that the direct overflow is independent from the size of the tree.[1]

We can also see by observing the results in Table 2, that the experimental results are consistent with formula (9). When using the formula for 1,00,000 access requests the number of accumulated blocks in the root is 10 and when the number of the access requests is 1,000,000 the formula gives 11. Finally, when using formula (9) for 10,000,000 access requests,

[1] A tree with L = 12 and N = 4096 and a tree with L = 13 and N = 8192. We obtained the same results as when we used a small tree with L = 10 and N = 1024.

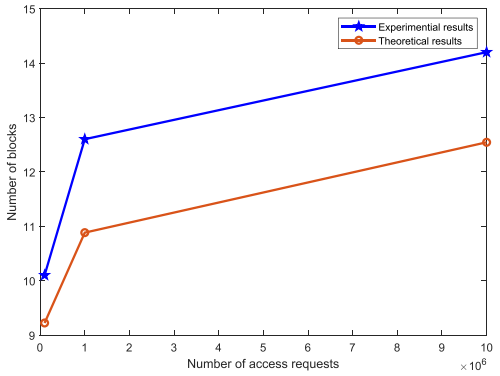

FIGURE 1. A comparison between the theoretical and experimental average direct overflow.

we get 13. A comparison between the experimental and theoretical results of the average direct overflow is displayed on Figure 1.

From Figure 1, we can see that the theoretical results are indeed lower than the experimental results as expected. Recall, that the mathematical model is optimistic and yields a lower root occupancy than that actually performed by the direct overflow. Indeed, while a sequence of RL requests interspersed by RR requests (respectively LR request interspersed by LL requests) is stopped when encountering the first LR or LL request (respectively the first RL or RR request), the accumulation of blocks at the root may continue. This takes place when the newly selected path is unable to host any block from the root. We observe on Figure 1 that the results given by the mathematical model are around 10% lower than the actual root direct occupancy values.

B. INDIRECT OVERFLOW FROM CHILDREN

As stated previously, the occupancy of the root bucket depends on the direct overflow as well as the indirect overflow from the children. Setting up the root bucket size to the sum of the maximum of the direct overflow and the maximum of the indirect overflow tacitly results in a very negligible failure probability. We believe that the number of overflowing blocks from children depends on z; the bigger is z the smaller is the number of overflowing blocks from the children and vice versa. Moreover, the number of access requests that are performed on the ORAM affects the number of overflowing blocks from the children. Increasing the number of access requests increases the probability of overflowing blocks.

Unfortunately, coming up with a closed form approximation as it was done for the direct overflow is not an easy task. To this end, we try to experimentally evaluate the indirect overflow by setting up a new Path ORAM with an infinite size root bucket only, and by fixing the size of the rest of the buckets to z. The used ORAM has height $L = 10$ and $N = 1024$. We vary the size of the buckets z from 1 to 5 and the number of access requests from 1,00,000, 1,000,000 and 10,000,000. We repeat each experiment ten times and calculate the CI

TABLE 3. Indirect overflow from children for infinite root bucket and a tree with N = 1024.

#	100,000			1,000,000			10,000,000		
z	**Av**	**Max**	**CI**	**Av**	**Max**	**CI**	**Av**	**Max**	**CI**
1	131	138	2.73	137.6	141	1.28	140.8	144	1.09
2	20.2	23	1.40	24.3	26	0.72	26.3	28	0.89
3	5.6	7	0.78	8.1	13	1.29	10.1	13	0.99
4	2.1	3	0.46	3.8	5	0.76	5.2	8	0.92
5	0.8	2	0.49	2.8	4	0.76	3	5	0.65

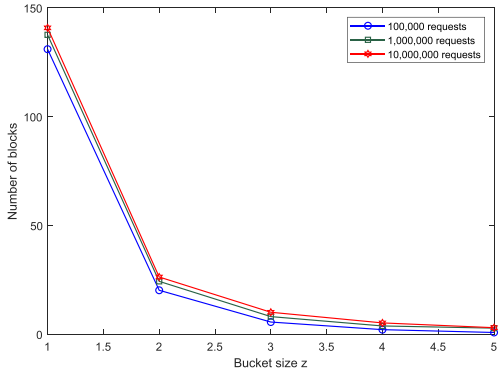

FIGURE 2. Indirect overflow for infinite root bucket and a tree with N = 1024.

at 95%. The obtained results are displayed in Table 3 and Figure 2.

We clearly observe that the indirect overflow depends on the bucket size z; the smaller is z the larger is the number of overflowing blocks. The indirect overflow is also affected by the number of requests; it gets larger for a larger number of requests. This is understandable as the indirect overflow is the maximum of the number of overflowing blocks, and the maximum of iid random variables depends on the number of these random variables as shown previously. However, we further observe from Figure 2 and especially from Table 3 that the indirect overflow is less sensitive to the increase of the number of access requests than to the bucket size z.

Moreover, we notice that the number of overflowing blocks from the children when $z = 5$ is very low 0.8, 2.8 and 3 respectively for the three considered numbers of access requests. This means that $Z = 5$ is enough to accommodate the blocks with negligible overflow and consequently a negligible probability of failure.

We also conducted experiments using a larger tree to ascertain the effect of the size of the tree on the indirect overflow. We experimented with a tree having $L = 12$ and $N = 4096$ and another larger tree having $L = 13$ and $N = 8192$. The results of the experiments are displayed in Table 4 and Figure 3 for the former and in Table 5 and Figure 4 for the latter.

When increasing the tree size, we notice that the number of overflowing blocks from the children into the root increases. This is understandable as when the tree size increases so does the number of blocks in the tree which enforces the impact of a smaller bucket size. The smaller is the bucket size the larger is the difference. For example, for $z = 1$ and for 1,00,000 access requests the number of

TABLE 4. Indirect overflow for infinite root bucket and a tree with N = 4096.

#	100,000			1,000,000			10,000,000		
z	Av	Max	CI	Av	Max	CI	Av	Max	CI
1	463.4	485	5.94	481.6	494	5.70	488	498	3.61
2	45	62	3.83	50.9	61	2.67	56.1	65	2.40
3	9.1	15	2.36	13.8	20	1.82	13.9	20	1.79
4	4	9	1.75	5.3	9	1.21	7.1	13	1.50
5	2	5	1.05	2.6	4	0.60	3.6	7	0.93

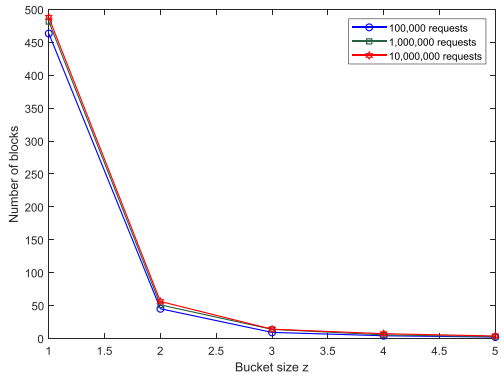

FIGURE 3. Indirect overflow from children for infinite root bucket and a tree with N = 4096.

TABLE 5. Indirect overflow from children for infinite root bucket and a tree with N = 8192.

#	100,000			1,000,000			10,000,000		
z	Av	Max	CI	Av	Max	CI	Av	Max	CI
1	878.9	894	6.44	911.9	935	7.16	925.9	943	5.26
2	71.5	86	3.80	83	91	2.37	87.2	96	2.94
3	10.3	13	0.96	15.3	21	2.24	17.3	21	1.30
4	3.1	5	0.70	4.6	7	0.63	6.1	8	0.58
5	0.9	3	0.70	2.7	5	0.62	3.5	6	0.64

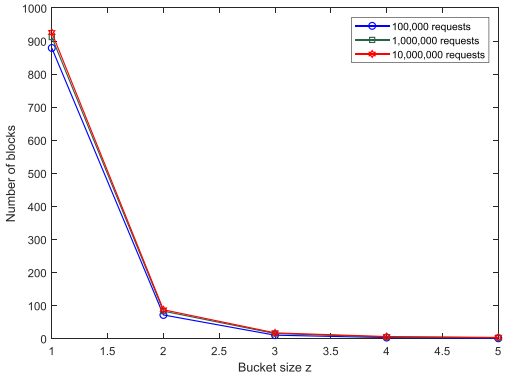

FIGURE 4. Indirect overflow from children for infinite root bucket and a tree with N = 8192.

overflowing blocks from the children into the root amounts to 131, 463.4 and 878.9 respectively for the trees with N = 1024, 4096 and 8192. However, using $z = 3$ amounts to an indirect overflow equal 5.6, 9.1 and 10.3 respectively for the trees with N = 1024, 4096 and 8192; the difference is much smaller here then it was when the bucket size was equal to 1.

VI. RADIX PATH ORAM

In view of the previous conducted analysis and obtained results, we propose a new version of Path ORAM with a large root (Radix) bucket size while using the same small z for all the other buckets in the tree. The root bucket size if set to the sum of the maximum of the direct overflow and the maximum of the indirect overflow, would necessarily guarantee a very negligible probability of failure. We name this new version the Radix Path (R-Path) ORAM. In the traditional Path ORAM, the de-facto size for z is 4, we propose to use smaller z and adjust the root size to an appropriate value able to enforce a negligible probability of failure. R-Path ORAM is then able to provide a tangible reduction in the storage at the server side, yet it amounts to a great gain in the average response time. The root bucket size in R-Path ORAM is much larger than that of the original Path ORAM, however it does not impact its efficiency should we use the root top caching to cache the root at the client side.

TABLE 6. Average response time and server storage size for R-Path ORAM with $z = 1, 2, 3$ and Path ORAM with $z = 4$.

z	1	2	3	4
Root size	157	41	26	4
Average response time	424.96	863.93	1296.78	1516.15
CI	0.18	0.21	0.32	0.35
Server storage	8380416	16760832	25141248	33538048

A. EXPERIMENTATION

We implemented R-Path ORAM for $z = 1, 2$ and 3. For a tree with L = 10 and N = 1024. The size of the root bucket for each z is calculated using the formula we developed (9) to estimate the direct overflow and the experiments for determining the indirect overflow from children. For example, for $z = 1$ by using formula (9) for 10,000,000 access requests we get 13, and by looking up table 3 and taking the maximum size of the root when the number of access requests is 10,000,000 gives us 144. Adding the two numbers gives 157 which is the maximum root size needed.

Six hundred random access requests are performed in each experiment. Experiments are repeated ten times to calculate the 95% CI. Recall that the average response time provided by the original Path ORAM, was 1516.15 ms. The obtained results for R-Path ORAM with $z = 1, 2, 3$ are displayed in the three first columns of Table 6. The fourth column of Table 6 represents the results for the seminal Path ORAM with $z = 4$. The average response time of an access request to a data block is defined as the time elapsing from the instant of issuing the request until the delivery of the data block to the user. The average response time provided by the original Path ORAM is 1516.15 ms as indicated in Table 6.

We clearly observe the tangible gain in the average response time; the smaller is z the higher is the gain. When using a bucket size $z = 1$, the average response time is reduced to 424.96 ms which amounts to almost a quarter of the average response time of the original Path ORAM. This is indeed a natural result since the number of blocks in each bucket was reduced to a quarter of its original size. This reduction leads to a decrease in the number of blocks read and written back in each ORAM access leading to a large reduction in the average response time. As for the storage

needed at the server side, it is reduced to 25%, 50% and 75% of the original size needed for Path ORAM, when using $z = 1$, 2 and 3 respectively.

VII. REDUCTION OF THE ROOT BUCKET SIZE

We introduce here a background eviction technique into R-Path ORAM to fulfill two objectives. The first is to ensure that the root bucket does not exceed its pre-defined capacity; and therefore we remove the possibility of a failure, namely we nullify completely the probability of failure. The second objective is to lower the needed root bucket capacity. The background eviction technique is called whenever the root bucket becomes full and serves to evict some blocks from the root bucket downward. Essentially, the background eviction is performed by issuing one or more requests for random data blocks. These requests are hereafter called dummy requests since they are different from the real read/write requests. A dummy request for a random data block amounts to reading the complete path (all the blocks of all the buckets on this path) from the root to the leaf to which the data block is mapped as stated in the position map. Then this path is written back using the greedy filling algorithm which pushes the data blocks downward the path as much as possible, which in turn may move some data blocks form the root downward the path. Note here that there is no re-mapping of the randomly accessed blocks. The question naturally arises as how to be sure that the background eviction removes (by pushing data blocks downward the tree using the greedy filling) some data blocks from the full root bucket. Our goal here is not to propose the best background eviction scheme, rather we want to show that the background eviction as a technique fulfils both of our mentioned objectives.

A background eviction technique should not breach the security of the ORAM and must ensure its obliviousness. In addition, an eviction technique should only add a small overhead in terms of additional storage and communication. As the technique we are proposing is based on dummy requests then: 1) It does not breach the security of the ORAM and does maintain its obliviousness since the dummy requests concern randomly chosen data blocks, yet dummy requests are handled exactly the same way as normal requests. The server or an adversary has no way of distinguishing between normal and dummy requests. It is true that in a dummy request for a random data block there is no re-mapping of the data block at the client side; but this action is transparent to the server and to any adversary, 2) there is no additional storage required either at the server or at the client and 3) as each dummy request amounts to the reading and the writing back of a complete path using the greedy filling, the proportion of dummy requests should be kept very low. The communication overhead of the eviction technique is proportional to the number of calls multiply the number of dummy requests per call. The background eviction is called each time the root bucket gets full. As such, a natural tradeoff builds up between the number of data blocks to remove from the root bucket at each call and the corresponding communication overhead.

TABLE 7. TWE technique for bucket size $z = 1$, root bucket size = 120, 130, 140 and 150 and number of access requests = 10,000,000.

Root bucket size	120	130	140	150
Number of calls	36725	5109	401	8
Number of evicted paths	73450	10218	802	16
Number of evicted blocks	51934	7415	578	11

There is also a natural tradeoff between the reduced deployed capacity of the root bucket and the frequency of calling the background eviction. The smaller is the defined capacity of the root bucket, the more frequent the background eviction is to be called.

A. THE TWO WAY EVICTION TECHNIQUE

The Two Way Eviction (TWE) technique just performs two dummy requests per eviction call. The first dummy request concerns a randomly selected data block from the left half of the ORAM tree and the second dummy request concerns a randomly selected data block from the right half of the ORAM tree. The rationale behind selecting a random data block (equivalently path) from each half of the tree is to solve the case where blocks accumulate in the root from direct overflow. In the case where blocks have accumulated in the root bucket from successive RL access requests, a dummy eviction request to the left half of the tree will most likely evict some blocks from the root bucket. On the other hand, if the blocks have accumulated in the root bucket from successive LR access requests a dummy eviction request to the right half of the ORAM tree will most likely evict some blocks from the root bucket. As we do not know which successive access requests led to the accumulation of blocks in the root bucket, we perform both dummy requests per eviction call.

B. EXPERIMENTATION WITH TWE

We implemented and integrated the TWE background eviction technique into the R-Path ORAM. We used ORAMs with L = 10, N = 1024 and performed 10,000,000 access requests per experiment. The experiments are divided into three groups. The first group is used to experiment with bucket size $z = 1$ and root bucket size equal 120, 130, 140, 150, 160. The second group of ORAMs is used to experiment with bucket size $z = 2$ and root bucket size 10, 12, 16, 20, 30 and 40. Finally, the third group of ORAMs is used to experiment with bucket size $z = 3$ and root bucket size 10, 12, 15, 20, 25.

During the experiments, we kept track of the number of calls made to the background eviction technique, as well as the cumulative number of evicted (removed from the root bucket) blocks. The number of dummy requests (equivalently the number of evicted paths) is readily equal to the double of the number of calls. The larger is the number of evicted paths the bigger is the communication overhead introduced by the background eviction technique.

TWE is never called when using the large root size; specifically, when using root size = 160 with $z = 1$, root

size = 40 with $z = 2$ and root size = 25 with $z = 3$. This is a natural result since the number of blocks in the root bucket never reaches the root size. An estimation of the maximum root size for each ORAM is readily given using formula (9) and Table 3. For a number of access requests of 10,000,000, formula (9) gives 13 and Table 3 gives 144, 28 and 13 respectively for $Z = 1, 2, 3$. Then adding the direct overflow to the indirect overflow from children readily gives the root sizes 157, 41 and 26 for bucket sizes $z = 1, 2$ and 3 respectively. For bucket size $z = 2$ and 3 even though we set the root size to be less than the maximum estimated number of blocks, during the run of the experiments the maximum number of blocks accumulated never exceeded the capacity of the root bucket resulting in zero calls.

The results of using the TWE technique with bucket size $z = 1$, root bucket sizes 120, 130, 140 and 150 are displayed in Table 7. For a root bucket size 150, the number of evicted paths (equivalently the number of dummy requests) is indeed extremely small and equals 16. As we reduce the root bucket to 120, the number of evicted paths grows and reaches 73450, but still this is just 0.74% of the total number of data access requests. This can be easily tolerated without affecting the request average response time, specially since the dummy evictions performed by the TWE technique are identically distributed among the data access requests. To sum up, we can reduce the root bucket size from 157 to 120 and the system incurs only 0.74% overhead defined as the proportion of additional dummy requests given we are performing 10,000,000 data access requests. We also note that the average number of evicted data blocks from the root is equal to around 1.42 blocks per eviction call when the bucket size $z = 1$. We would like to note here that while $z = 1$ represents the extreme case of R-Path its corresponding root bucket size can be successfully reduced by the TWE background eviction technique.

To maintain a negligible failure probability, the traditional Path ORAM, in the case of L = 10 and $Z = 4$, requires a stash of capacity equal to 133 blocks: 89 extra blocks to enforce a negligible failure probability [14] and 4*11 = 44 blocks for the retrieved path. On the other hand, for the extreme case of z = 1, the proposed Radix-Path ORAM requires a client storage of 167 blocks: 157 blocks for the root and 10*1 = 4 10 blocks for the retrieved path. This amounts to an additional storage of 34 blocks. Furthermore, the use of our proposed TWE technique may reduce the storage needed at the root from 157 to 120 with virtually a negligible impact on the average delay. To sum up, even for the extreme case of z = 1, the proposed R-Path ORAM with the root top caching and the TWE technique outperforms the seminal Path ORAM with z = 4. Furthermore, since we are targeting the domain of cloud storage and not that of secure processors, a small additional client storage should not be that problematic as even cheap smart mobile phones are currently equipped with few GB of storage.

The results of using the TWE technique with bucket sizes $z = 2$ and 3 are displayed in Tables 8 and 9. First of all, we clearly observe that the number of evicted paths decreases with the increase of the bucket size z. This is understandable as for a larger z a path in the ORAM tree accommodates a higher number of blocks and consequently the pressure on the root bucket is reduced. The results achieved by the TWE technique are impressive. For example, when $z = 2$ and the root bucket size = 20 the number of evicted paths is 22860. That is, we can save 50% on the storage of the root bucket size (i.e. recall that for $z = 2$ maximum root bucket size = 41) by introducing an overhead of just 0.23% of dummy evictions. Moreover, using $z = 3$ provides even better results, for example when the root bucket size equals 15, the number of evicted paths is only 514. This means that we can save 50% on storage needed for the root bucket by injecting a tiny overhead of 0.005% dummy requests.

TABLE 8. TWE technique for bucket size $z = 2$, root bucket size = 10, 12, 16, 20, 30 and number of access requests = 10,000,000.

Root bucket size	10	12	16	20	30
Number of calls	309539	179238	52210	11430	107
Number of evicted paths	619078	358476	102420	22860	214
Number of evicted blocks	544370	333752	104444	25253	276

TABLE 9. TWE technique for bucket size $z = 3$, root bucket size = 10, 12, 15 and 20 and number of access requests = 10,000,000.

Root bucket size	10	12	15	20
Number of calls	6395	1669	257	6
Number of evicted paths	12790	3338	514	12
Number of evicted blocks	19203	5240	759	26

VIII. CONCLUSION

R-Path ORAM is an improved version of the seminal Path ORAM. R-Path ORAM allows to use a small bucket size for all buckets in the ORAM binary tree but the root bucket which uses a large bucket size. First, we conducted a detailed analysis of the root bucket required size and then we developed a mathematical model providing a closed form expression of the root bucket size. Reducing the bucket size tacitly amounts to a tangible gain in the server storage as well as in the request average response time. Extensive experiments were conducted on a developed platform to ascertain the efficiency of R-Path and its gain in server storage and request average response time.

We also proposed the use of a background eviction scheme to reduce the root bucket size and to nullify the probability of failure. We proposed the Two Way Eviction scheme that uses just two clever dummy requests per eviction call. Conducted experiments showed that our proposed TWE scheme presents indeed a very efficient technique to reduce the needed root bucket size, yet completely prune any possibility of the system failure due to an overflow of the root bucket.

While the proposed TWE is very efficient, further investigation is needed to find the optimal background eviction technique that provides the largest reduction of the root bucket size for the smallest overhead in terms of the proportion of the added dummy requests.

REFERENCES

[1] *The Treacherous 12: Cloud Computing Top Threats in 2016*, Top Threats Working Group, Cloud Secur. Alliance, Seattle, WA, USA, Feb. 2016. [Online]. Available: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwjG4tWh45bjAhUJWZoKHc3zAxQQFjAAegQIAxAC&url=https%3A%2F%2Fdownloads.cloudsecurityalliance.org%2Fassets%2Fresearch%2Ftop-threats%2FTreacherous-12_Cloud-Computing_Top-Threats.pdf&usg=AOvVaw3fIsG4hw7kKfeHPAinL0ps

[2] C. Sahin and A. El Abbadi, "Data security and privacy for outsourced data in the cloud," in *Proc. EDBT*, Apr. 2017, pp. 606–609.

[3] R. Bost, "Algorithmes de recherche sur bases de données chiffrées," Ph.D. dissertation, Univ. Rennes 1, Rennes, France, Jan. 2018.

[4] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Proc. NDSS*, vol. 20. 2012, p. 12.

[5] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 18:1–18:51, Aug. 2014. doi: 10.1145/2636328.

[6] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proc. 19th Annu. ACM Symp. Theory Comput.*, Jan. 1987, pp. 182–194. doi: 10.1145/28395.28416.

[7] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[8] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," 2011, *arXiv:1106.3652*. [Online]. Available: http://arxiv.org/abs/1106.3652

[9] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost," in *Advances in Cryptology—ASIACRYPT*, D. H. Lee and X. Wang, Eds. Berlin, Germany: Springer, 2011, pp. 197–214.

[10] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound," in *Proc. 22Nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 850–861. doi: 10.1145/2810103.2813634.

[11] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Ring ORAM: Closing the gap between small and large client storage oblivious ram," *IACR Cryptol. ePrint Archive*, vol. 2014, p. 997, Aug. 2014.

[12] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A constant bandwidth blowup oblivious RAM," in *Proc. Theory Cryptogr. Conf.*, 2016, pp. 145–174.

[13] C. W. Fletcher, "Oblivious ram: From theory to practice," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2016.

[14] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious ram protocol," *J. ACM*, vol. 65, no. 4, pp. 18:1–18:26, Apr. 2018. doi: 10.1145/3177872.

[15] B. Li, Y. Huang, Z. Liu, J. Li, Z. Tian, and S.-M. Yiu, "HybridORAM: Practical oblivious cloud storage with constant bandwidth," *Inf. Sci.*, vol. 479, pp. 651–663, Apr. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025518301002

[16] M. Sánchez-Artigas, "Enhancing tree-based ORAM using batched request reordering," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 3, pp. 590–604, Mar. 2018. doi: 10.1109/TIFS.2017.2762824.

[17] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: A dissection and experimental evaluation," in *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, 2016.

[18] P. Teeuwen, "Evolution of oblivious ram schemes," M.S thesis, Dept. Math. Comput. Sci., Technische Univ. Eindhoven, Eindhoven, The Netherlands, 2015.

[19] K. Al-Saleh and A. Belghith, "Practical suitability and experimental assessment of tree orams," *Secur. Commun. Netw.*, 2018. doi: 10.1155/2018/2138147s.

[20] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 299–310. doi: 10.1145/2508859.2516660.

[21] L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious RAM in secure processors," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 571–582, 2013.

[22] C. W. Fletcher, M. V. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proc. 7th ACM Workshop Scalable Trusted Comput.*, Oct. 2012, pp. 3–8. doi: 10.1145/2382536.2382540.

[23] C. W. Fletcher, "Ascend: An architecture for performing secure computation on encrypted data," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2013.

[24] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "PrORAM: Dynamic prefetcher for Oblivious RAM," in *Proc. 42Nd Annu. Int. Symp. Comput. Archit.*, Jun. 2015, pp. 616–628. doi: 10.1145/2749469.2750413.

[25] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di, "Fork path: Improving efficiency of oram by removing redundant memory accesses," in *Proc. 48th Int. Symp. Microarchitecture*, Dec. 2015, pp. 102–114. doi: 10.1145/2830772.2830787.

[26] K. AlSaleh and A. Belghith, "Locality aware Path ORAM: Implementation, experimentation and analytical modeling," *Computers*, vol. 7, pp. 1–19, Oct. 2018.

[27] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proc. 2013 ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2013, pp. 311–324. doi: 10.1145/2508859.2516692.

[28] B. Eisenberg, "On the expectation of the maximum of iid geometric random variables," *Statist. Probab. Lett.*, vol. 78, pp. 135–143, Feb. 2008.

[29] I. Gradshteyn and I. Ryzhik, *Table of Integrals, Series, and Products*. New York, NY, USA: Academic Press, 1980.

KHOLOUD SAAD AL-SALEH received the M.S. degree in information systems and the Ph.D. degree in computer science from King Saud University, in 2002 and 2019, respectively, where she is currently an Assistant Professor with the Department of Information Technology, College of Computer and Information Sciences. Her research interests include computer networks, cloud computing, edge computing, and network and cloud security issues. She is also a member of the IEEE Communications Society.

ABDELFETTAH BELGHITH received the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles (UCLA), in 1982 and 1987, respectively. Since 1992, he has been a Full Professor with the National School of Computer Sciences (ENSI), University of Manouba, Tunisia. He is currently on a sabbatical leave at King Saud University, Saudi Arabia. He is also the Director of the HANA Research Laboratory, National School of Computer Sciences. He has published more than 350 research papers in international journals and conference proceedings. His research interests include computer networks, wireless networks, the multimedia Internet, mobile computing, distributed algorithms, systems and information security, and simulation and performance evaluation. He runs several research projects in cooperation with other universities, research laboratories, and research institutions. He is also the past Chair of the IEEE Tunisia Section and the Chair of the IEEE ComSoc and VTS Tunisia Chapters.