computers

Article

Locality Aware Path ORAM: Implementation, Experimentation and Analytical Modeling

Kholoud Al-Saleh and Abdelfettah Belghith *

College of Computer and Information Sciences, King Saud University, Riyadh 11451, Saudi Arabia; kholoud_alsaleh@yahoo.com

* Correspondence: abelghith@ksu.edu.sa; Tel.: +966-535-920-540

Received: 10 October 2018; Accepted: 25 October 2018; Published: 29 October 2018

Abstract: In this paper, we propose an advanced implementation of Path ORAM to hide the access pattern to outsourced data into the cloud. This implementation takes advantage of eventual data locality and popularity by introducing a small amount of extra storage at the client side. Two replacement strategies are used to manage this extra storage (cache): the Least Recently Used (LRU) and the Least Frequently Used (LFU). Using the same test bed, conducted experiments clearly show the superiority of the advanced implementation compared to the traditional Path ORAM implementation, even for a small cache size and reduced data locality. We then present a mathematical model that provides closed form solutions when data requests follow a Zipf distribution with non-null parameter. This model is showed to have a small and acceptable relative error and is then well validated by the conducted experimental results.

Keywords: oblivious RAM; Path ORAM; data locality; Zeta function; caching; performance evaluation

1. Introduction

Information technology has revolutionized many aspects of everyday life. One major new technology, which facilitated the goal of delivering computing as a utility, is cloud computing. It is envisioned that cloud computing will play a pivotal role in storage, social networking and delivery of multimedia systems. However, cloud computing brings along many challenges, specifically in terms of security [1]. The growth rate of data stored on the cloud is huge. Client-side encryption has been used to provide security to the data stored in the cloud. However, client-side encryption alone is not enough to provide security to data stored on the cloud, since an adversary can observe the access pattern to the stored data and can discover a lot of information from this access pattern [2]. Fortunately, there is a cryptographic primitive that hides the access pattern. This cryptographic primitive is called Oblivious Random Access Memory (ORAM) [3].

ORAM was initially developed by Goldreich and Ostrovsky [3–5]. The basic idea of ORAM is to randomly permute and shuffle the data stored in memory, so that no data block is stored in the same position again and no relation between the accessed data blocks can be deduced by the adversary. Any ORAM scheme must use probabilistic encryption to encrypt the data before it is stored on the server. Then when the data is either read or written by the user, the user must decrypt it first, perform a task with it, then the data must be encrypted again with probabilistic encryption before it is sent back to the server again.

The first (obvious) ORAM had to read the whole memory to perform a single read or write, and write the whole memory back again. Moreover, when the user receives the whole content of the memory they must decrypt it then encrypt it again before it can be written back to the server. This ORAM incurs an enormous overhead in bandwidth and processing time, which prevents any practical use of it. Fortunately, this first trivial ORAM was followed up by many ORAM constructions and improvements [6–17]. The basic idea in these new ORAMs, to make the data access oblivious, is to introduce an extra storage at the server and continuously and obliviously shuffle the data so that no data block is saved in its same previous location. The continuous shuffling ensures the obliviousness of the access pattern.

Current ORAM constructions can be grouped into two main groups: Hierarchical ORAM and Tree ORAM [18]. There are many ORAM constructions and variants in each group. However, Tree ORAMs are the most recent and have better performance than Hierarchical ORAM. This is essentially due to the fact that Hierarchical ORAMs require expensive oblivious shuffling and sorting operations.

There is a number of tree ORAM constructions among them are: Path ORAM [12,19], Ring ORAM [15], XOR Ring ORAM [15,20] and Onion ORAM [16]. However, Path ORAM is the most practical as a result of its simplicity [19]. In this paper, we will concentrate on Path ORAM and try to further enhance its performance and implementation to take advantages of any popularity in the requested data.

ORAMs in general do not protect against a kind of information leakage that can be obtained by observing the timing of operations generally referred to as the timing channel attack [19,21]. Path ORAM like most ORAMs tacitly ignores the timing channel attack [19].

Most of real-world applications show an immense rate of data locality and popularity. Accessing outsourced data is no exception. The popularity of a data document in a collection of documents is the number of times the document is requested compared to the number of times the rest of the documents in the collection are requested in a certain period of time [22]. On the other hand, data locality may also refer to data parts or segments within the same document. Some parts of a document are accessed much more often than other parts of the same document. For example, the abstract of a research paper has data locality higher than other parts of a research paper. This kind of data locality is referred to as temporal data locality. Spatial locality, that is accessing data blocks close to each other, is very important in the secure processor setting [23–25]. However, the ORAM literature has overlooked data locality and popularity. Using ORAM inherently destroys any spatial data locality, since ORAM depends on storing the data in random locations. As such, we will only concentrate on temporal locality and popularity to enhance the performance of Path ORAM.

Path ORAM has gained popularity in the research community due to its good performance and its simple and elegant algorithm. Many work have tried to enhance its performance specially in the secure processor domain. However, to the best of our knowledge, there is no work in the secure storage outsourcing domain that has improved the Path ORAM performance using the popularity or locality of the requested data. Data accessed by the user tend to exhibit some form of popularity and locality that can be used to enhance the performance of any ORAM. We show in this paper that the addition of some small storage at the client side allows to take advantage of the existence of any data popularity and locality. This will amount to a much better response time compared to that of the original Path ORAM.

Our contributions are the following:

1. Enhancing the implementation of Path ORAM by integrating the necessary awareness about any existent popularity within the sequence of requested data blocks
2. Showing the superiority of the enhanced implementation even for a small additional storage and a reduced data locality
3. Developing a mathematical model providing adequate closed form solutions for the size of the extra storage as well as the hit ratio for any given non-null popularity parameter
4. Validating our proposed mathematical model using the conducted experimental results.

The rest of the paper is organized as follows. We present a detailed description of Path ORAM in Section 2. In Section 3, we present the relevant related work to our research. Then in Section 4, we present the enhancement and modification we propose to get advantage of data popularity. In Section 5, we first describe the test bed used for the experiments, then we present and interpret the results of our conducted experiments. In Section 6, we present the mathematical model based on the

Zipf distribution. This provides closed form approximated solutions for the cache size and cache hit. Our proposed mathematical model is then validated using the results obtained from the conducted experiments. Finally, in Section 7, we present some concluding remarks.

2. Path ORAM

Stefanov et al. [12] introduced a very simple and elegant ORAM, named Path ORAM. Path ORAM uses a binary tree structure just like the initial Tree ORAM proposed by Shi et al. [9] to store the data on the server. The binary tree is composed of constant size nodes called buckets. To store N data blocks the tree has to have N buckets. Each bucket can store up to some constant number z of data blocks [12,26,27]. If the number of data blocks in a bucket is lower than the constant number z then the bucket must be filled up with dummy blocks until the total number of blocks in the bucket reaches z.

Every data block in Path ORAM is mapped to a random leaf; this information is stored in a position map stored at the client. When a data block is mapped to a leaf it means that it must be stored in a bucket residing on the path from the root of the tree to that leaf. The leaves are numbered starting from zero. The client also has a buffer reserved to store the read data blocks. This buffer is named a stash. The stash must be able to hold at least a complete path read from root to leaf. Thus, the size of the stash must be at least $z \log(N)$, where N is the number of buckets in the binary tree. Path Oram has the following invariant: a data block must be stored on the path it is mapped to or be in the stash.

The notations used for Path ORAM are displayed in Table 1 and the read-write algorithm is displayed in Algorithm 1. To read/write a data block, the position map has to be looked up first to find the leaf the data block is mapped to and stored in the variable x line 1 in the algorithm. Following that, the data block must be mapped to a new random leaf and the position map has to be updated to reflect this change line 2 in the algorithm. Then all the blocks residing in the buckets along the path from the root to leaf x are read and decrypted then stored into the stash lines 3, 4 and 5 in the algorithm. Storing only real data blocks and discarding any dummy blocks. If the operation is a write the content of the data block will be changed to the new content lines 6, 7 and 8 in the algorithm. Then the path just read has to be written back to the server. This step is done using a greedy filling strategy starting from the leaf and filling up buckets on the path with data blocks from the stash that satisfy the invariant of Path ORAM. Each bucket on the path must hold z blocks if there are not enough blocks in the stash to fill up the bucket, then the bucket will be filled up with dummy blocks lines 9–15 in the algorithm. Finally the data is returned to the user line 16 in the algorithm.

We must here note that before sending any block whether a data block or a dummy block, it must be encrypted first [12,28].

Table 1. Symbols used for Path ORAM.

Symbol	Meaning
$\boldsymbol{N}$	The number of buckets
$\boldsymbol{L}$	Tree height
$\boldsymbol{z}$	Size of a bucket in blocks
operation	Read or Write represented by R or W
$\boldsymbol{a}$	The address of the data block to be read or written
data*	This field holds the data if the operation is a Write and empty otherwise
$\boldsymbol{P(x)}$	Path from root to leaf node x
$\boldsymbol{P(x, i)}$	Bucket at level i on path x
$\boldsymbol{St}$	The client’s local stash
$\boldsymbol{PosM}$	Position map
$\boldsymbol{x := PMap[a]}$	Block a is assigned to leaf node x
$\boldsymbol{RB(i)}$	Read a whole bucket i
$\boldsymbol{WB(i, y)}$	Write y into bucket i
$\boldsymbol{UR}$	Uniform Random distribution

Algorithm 1 Algorithm for data access of Path ORAM

```
Input: operation, a, data*
1: $x \leftarrow PosM[a]$
2: $PosM[a] \leftarrow UR(0...2^L - 1)$
3: for $i = 0$ to $L$ do
4:     $St \leftarrow (St \cup RB(P(x,i))$
5: end for
6: if $(operation = W)$ then
7:     $St \leftarrow (St - (a, data)) \cup (a, data^*)$
8: end if
9: for $i = L$ downto 0 do
10:     $St \leftarrow St \cup RB(P(x,i))$
11:     $St' \leftarrow (a', data) \in St : P(x,i) = P(PosM[a^{'}], i)$
12:     $St' \leftarrow Minmum(|St^{'}|, z)$ blocks from $St'$
13:     $St \leftarrow St - St'$
14:     $WB(P(x,i), St')$
15: end for
16: return data
```

3. Related Work

Ren et al. [29] introduced three techniques to optimize the performance of Path ORAM in the secure processor setting. The first technique is background eviction that allows to decrease the number of blocks z stored in each bucket. This amounts in decreasing the overhead as well as lowering the failure probability. When the number of blocks in the stash reach a certain threshold, background evictions are performed by issuing dummy read requests. Background evictions are kept oblivious and cannot be distinguished from normal ORAM access. The second technique in the secure processor setting is static super block. In this technique, blocks that exhibit some form of locality are grouped into a super block and assigned to the same leaf in the Path ORAM tree. Thus, when one block is read, all the other blocks belonging to the same super block are also read. Furthermore, after reading a block in a super block, not only that block gets assigned to a new random leaf but all the blocks belonging to the same super block get assigned to the same random leaf. In their work, they used address space locality to group the blocks of a program into super blocks, and this grouping had to be done before loading the ORAM tree and was static. As such, their work is limited to program locality and not data locality. The third technique is subtree layout where they proposed packing subtrees with k levels together. These subtrees are handled as the buckets of a new tree [29].

Fletcher et al. developed a secure processor that uses Path ORAM and defeats the timing channel attack by imposing the access to the Path ORAM at a fixed predetermined rate. This rate is determined offline. Thus, if a request is present before the next allowed request time, it must wait. Moreover, if there is no request at the allowed request time, a dummy request must be sent. This solution clearly degrades the performance of the ORAM and introduces a large amount of overhead. Fletchery et al. [30] showed that this secure processor has more than 50% overhead in power and performance. They proposed a dynamic scheme that allows a small amount of leakage. This scheme reduces the performance deterioration by 30% [30].

Maas et al. [31] introduced the first Path ORAM implementation on hardware. They named it Parallel Hardware to make Applications Non-leaky Through Oblivious Memory (Phantom). Two techniques were used to enhance the performance of Path ORAM in Phantom. The first technique is, treetop caching. In tree top caching the first k levels of the tree are saved in the stash so that when reading/writing to the ORAM only the lower layers are updated reducing the latency and overhead. The second technique used in Phantom is min-heap eviction where the stash is stored as a min-heap evicting the blocks that are least recently used first.

Yu et al. [32] improved on the previous work of Ren et al. [29] that used static super block and proposed a dynamic super block approach that allows the contents and size of the super block to change during the run of the ORAM taking into consideration the programs locality. They called their new approach Dynamic Prefetcher for ORAM (PrORAM). However, this PrORAM is only for secure processor and programs and not for outsourced data.

Ren et al. [33] introduced Unified ORAM. Basically, Unified ORAM, is a recursive Path ORAM that has been updated to allow for the use of locality in the position map and pseudorandom compression to store the position map. Recursion is used in ORAMs to reduce the storage needed for the position map at the client. It is more important for ORAMs implemented on hardware due the constraints of on-chip area. The main objective of their work was to reduce the overhead introduced by recursive Path ORAM. They did so by using the locality in the position map. Moreover, they were able to reduce the size of the position map to be stored on-chip by using some pseudorandom compression. However, their work only takes into consideration the locality of the position map and not the locality of the data itself.

Fletcher et al. [34] developed three techniques to improve the performance of any recursive ORAM. Their work is an enhancement to the work of Ren et al. [33]. They proposed a Position map Lookaside Buffer (PLB) that uses the locality of the position map, in addition to using compression methods for storing the position map. Moreover, they introduced a Position map Message Authentication Code (PMAC) to ensure integrity and verification. However, their work is restricted to recursive ORAM and is more important for ORAMs implemented in hardware. Furthermore, the locality used is only the locality of the position map and not the locality of the data itself.

Zhang et al. [35] introduced Fork Path ORAM. Fork Path ORAM merges two consecutive ORAM requests together. They tried to make use of the fact that two consecutive requests might have overlapping buckets in their paths. Thus, they suggested when a read/write request is performed and the whole path of the desired data block is read from the server and loaded into the stash, to wait and postpone the writing back of the complete path until having the subsequent request. When the subsequent request comes, the buckets that overlap in the two paths are not written back, and only the buckets that are in the path of the first request are written back. Moreover, to process the second request only the buckets in the second path that do not intersect with the path of the first request are read into the stash. Since the overlapping buckets have been fetched and read previously by the first request. Then the process is continued with the second request and third request and so on. They further suggested the rescheduling of pending ORAM requests. However, all their work was with the secure processor setting, yet the benefit of the merging of the requests is minimal as showed by Sanchez [36]. Sanchez showed that grouping requests of size two can achieve a saving of one bucket (i.e., only one bucket does not have to be re-read, which is the root). While increasing the grouping of requests from two to five can only achieve a saving of 2.25 buckets.

Sanchez [36] proposed merging requests in groups of more than two and working in batches. Furthermore, he proposed to dynamically reorder the batch requests to achieve the maximum overlap between paths of a batch. Unfortunately, this work depends on a very hard assumption of having all the requests in advance. This assumption is not practical at all for outsourced data. They showed in their experiments that grouping five requests together can only achieve a saving of 2.25 buckets. This saving is in the top levels of the tree (i.e., the root and the two levels below it.) These top three levels only contain seven buckets. We could have just read the first three levels and saved them in the stash by using the tree-top caching technique proposed by Maas et al. [31]. Moreover, the problem was formulated as a partition problem making it an NP-hard problem.

Asharov et al. [18] stated that previous work on ORAMs ignored locality and that they are the first to take locality into consideration when designing ORAMs in the context of secure processors.

They argue that in many reasonable applications a user accessing neighboring memory locations is not kept a secret. Like in applications where the user asks for information between specific dates.

Chakraborti et al. [37] designed a hierarchical ORAM that uses locality however, this ORAM is a write-only ORAM based on the work done by Li et al. [38]. A write only ORAM preserves only the security of write operations and not that of read operations. Read operations are treated normally and their obliviousness is not preserved.

4. Path ORAM Enhancement

In the quest to improve the performance of Path ORAM and make it able to take advantage of data popularity/locality, we propose using a cache of reduced size at the client side to capture any eventual popularity/locality in the sequence of requested data blocks. As such, for each read request issued to the ORAM, we first check the cache if it contains the requested data block. In the affirmative, this will free the ORAM from having to fetch the data from the server. All write requests, on the other hand, have to be fetched from the server and then written back, but they are also stored in the cache according to their level of popularity/locality. A cache replacement policy must be used to manage our extra storage and segregate between stored data blocks according to their level of popularity. Cache replacement policies have been extensively studied in the context of the web caching. There are many replacement policies that have been proposed. Different policies suite different environments [39]. However, it is well known that three policies in general outperform the rest of the policies. The first policy is Greedy Dual Size Frequency (GDSF). Unfortunately, this policy works only for variable block size [40], and is consequently not applicable in our case of fixed size data blocks. The second policy is the Least Frequently Used (LFU) replacement policy where the least frequently accessed data blocks are replaced first. The third replacement policy is the Least Recently Used (LRU) where the least recently accessed data blocks are replaced first [41]. It should be stressed here that our main objective is to extend Path ORAM to make it capable of taking advantage of any eventually data popularity and not to propose the best replacement strategy. However, it remains important to be able to lower at most the extra storage (i.e., the size of the local cache) to be added locally at the client. To this end, we shall develop a mathematical model to ascertain the minimum capacity of our extra storage to satisfy a certain hit ratio; and this independently of the used replacement strategy.

The cache used does not have to be big in fact it can be the size of five data blocks. With the current advance in technology this is very affordable, even in mobile phones and personnel devices. We show the performance gain is huge and we implement this addition to Path ORAM in real experiments using varying cache sizes of five, ten, fifteen and twenty data blocks. Moreover, we vary the popularity/locality rate of the distribution of the requested data blocks. We show that even for data with a low popularity/locality rate the gain is still good. The gain can be measured using the hit ratio defined as the percentage of client requests satisfied by the cache without the need to access the server. The higher is the hit ratio the larger is the gain and the lower is the response time of our ORAM.

Usually data blocks accessed by the user are not uniformly distributed over the set of data blocks outsourced at the server. The Zipf distribution is often used to model a non-uniform access to a database [42–45]. The Zipf distribution is also known as the 80:20 or 90:10 law. This law states that most of (from 80 to 90%) the requests address a small batch (from 10 to 20%) of the population. With the Zipf law, the object with the highest frequency is selected twice as often as the object with the second highest frequency, then the object with the second highest frequency is requested twice as often as the object with the third highest frequency and so on.

Formally the truncated version of the Zeta distribution (i.e., the Zipf law) is:

$$P[X = r, \alpha] = \frac{r^{-\alpha}}{\sum_{i=1}^{N} i^{-\alpha}} \qquad \alpha > 0 \tag{1}$$

where r represents the rank of the requested data block, N the total number of data blocks, and α the Zipf parameter (also called the popularity parameter). $P[X = r, \alpha]$ is then the probability of the requested block to be of rank r given the parameter α, it also represents the frequency of blocks of rank r. We denote the harmonic number of order N and α by $H_N(\alpha)$; that is:

$$H_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha} \tag{2}$$

The parameter α ($\alpha > 0$) plays a major role in the Zipf distribution as it determines the shape of the cumulative probability function and regulates the deterioration in requests frequencies [42,45–48]. This parameter α can have different values for different applications. When α takes a different value than 1, the distribution is usually called Zipf-like distribution. Virtually in all previous research studies on web caching, α was considered to be less or equal to one [49–51].

Recently new evidence showed that popularity has become more important than before, and values of $\alpha > 1$ are becoming more common due to the proliferation of the Internet of Things and social networking [40,52,53]. The popularity is more concentrated than before, and therefore, caching becomes a more profitable approach. In [52], the authors analyzed 14 websites and showed that all of them provided a value $\alpha > 1$.

The modification to the Path ORAM algorithm can be done in a very simple manner. When a read request is submitted by the user the algorithm first checks to see if the block is already in the cache. If the data block is in the cache, then there is no need to proceed with the Path ORAM algorithm and just return the data block to the user. However, if the block was not present in the cache the block needs to be fetched from the server using the Path ORAM algorithm. In either case, the cache will be updated using the selected replacement strategy.

It is important to note that the invariant of Path ORAM remains unchanged since any block is either in the tree or in the stash. The copy we are keeping in the cache is an extra copy that is local to the user and does not affect the Path ORAM algorithm or violate its security.

5. Experimentation

5.1. Test Bed Description

The platform we have set up is made of one server, one client and a network switch. The server has 8 cores, 1 TB hard disk and 128 GB of random access memory. The client is a laptop with Intel core i7-7 500 u, up to 3.5 GHz with 1 TB hard disk and 8 GB of memory. The switch used to connect the client and the server has a speed of up to 1.09 Gbps. Ubuntu 14.04 is used at both the server and the client. The cloud server hosts a MongoDB instance as the outsourced cloud database and storage. The platform is displayed in Figure 1.

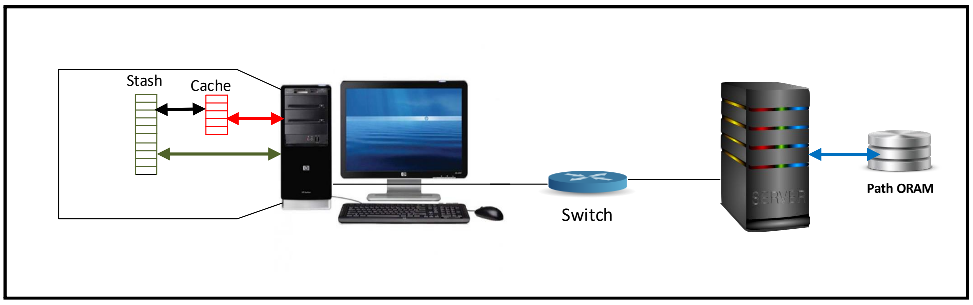

Figure 1. Experimental Platform.

We used the C++ programming language to implement the non-recursive version of Path ORAM. Moreover, AES/CFB was used for encryption with a key length of 128 bits. We set the block size to 4096 Bytes and the capacity of each bucket z to 4 data blocks. We set up an ORAM with N = 2047 buckets as well as real data blocks in the binary tree. We assumed that the issuing of block requests follows a Poisson distribution with an inter issuing time of one minute. The data block is, however, selected according to a Zipf distribution with parameter α among all real blocks in the ORAM.

We varied the skewness parameter α (i.e., the popularly rate) to show the amount of gain in performance that can be achieved. The values we used were: 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8 and 2. Moreover, we experimented with different cache sizes of five, 10, 15 and 20.

We ran the experiments until 600 requests were issued by the user.

Experimental Results, Assessment and Analysis

We performed 10 replications of each experiment to attain a 95% confidence interval. As such, we conducted 640 experiments in total where we varied the replacement strategy, the cache size and the popularity parameter (the value of α). We collected the experimental results for the request average response time and the hit ratio. The request average response time is defined as the average time from the instant the user issues a read/write request until the requested data is returned. The hit ratio is defined as the number of times the requested data block is found in the cache over the total number of block requests. The request average response time for the original Path ORAM (without using a cache) with link speed of 2 Mbps is 1514.64 ms. Obtained experimental results by varying the cache size from 5, 10, 15 and 20 data blocks, and using the following values 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8 and 2 for α are displayed in Tables 2–5 and sketched in Figures 2–5. We observe from the results that the LFU replacement policy gives better average response time and hit ratio than the LRU replacement policy. Moreover, achieving a minimum hit ratio of 25% only happens if the skewness parameter α of the distribution has a minimum value of 1. Smaller values of α only give very small hit ratios and thus a very small improvement in the average response time. For example, using $\alpha = 0.6$ and cache size of five blocks using the LRU and LFU replacement policies give an average response time of 1504.54 ms and 1455.40 ms respectively compared to the normal Path ORAM that gives 1514.64 ms.

Table 2. Average response time in ms for caches five, 10, 15 and 20 and different popularity rates using LRU cache replacement strategy and 2 Mbps link speed.

Popularity Rate	Cache 5	Cache 10	Cache 15	Cache 20
0.6	1504.54	1497.81	1475.93	1449.01
0.8	1449.01	1442.27	1396.83	1369.91
1	1353.08	1252.10	1171.32	1130.93
1.2	1110.74	928.98	848.20	787.61
1.4	848.20	626.05	545.27	525.08
1.6	605.86	363.51	323.12	262.54
1.8	403.90	242.34	201.95	158.20
2	262.54	141.37	112.14	103.50

Table 3. Hit ratio for caches five, 10, 15 and 20 and different popularity rates using LRU cache replacement strategy and 2 Mbps link speed.

Popularity Rate	Cache 5	Cache 10	Cache 15	Cache 20
0.6	0.01	0.01	0.03	0.04
0.8	0.04	0.05	0.08	0.10
1	0.11	0.17	0.23	0.25
1.2	0.27	0.39	0.44	0.48
1.4	0.44	0.59	0.64	0.65
1.6	0.6	0.76	0.79	0.83
1.8	0.73	0.84	0.87	0.89
2	0.83	0.91	0.92	0.93

Table 4. Average response time in ms for caches 5, 10, 15 and 20 and different popularity rates using LFU cache replacement strategy and 2 Mbps link speed.

Popularity Rate	Cache 5	Cache 10	Cache 15	Cache 20
0.6	1455.40	1442.27	1411.98	1405.81
0.8	1347.02	1297.54	1282.40	1228.54
1	1151.13	1029.96	989.57	928.980
1.2	868.39	747.22	666.44	626.05
1.4	646.25	484.69	426.02	424.10
1.6	444.30	323.12	262.54	242.34
1.8	262.54	181.76	161.56	151.46
2	141.37	121.17	100.98	100.98

Table 5. Hit ratio for caches 5, 10, 15 and 20 and different popularity rates using LFU cache replacement strategy and 2 Mbps link speed.

Popularity Rate	Cache 5	Cache 10	Cache 15	Cache 20
0.6	0.04	0.05	0.06	0.07
0.8	0.11	0.14	0.15	0.19
1	0.24	0.32	0.35	0.39
1.2	0.43	0.51	0.56	0.59
1.4	0.57	0.68	0.71	0.72
1.6	0.70	0.79	0.83	0.84
1.8	0.80	0.88	0.89	0.90
2	0.87	0.92	0.93	0.93

Furthermore, we can see that the LRU replacement policy is affected by the increase of the size of the cache more than the LFU replacement policy. For example, for α = 1.2 and changing the cache size from five to ten the LRU hit ratio increase by 12% on the other hand the LFU hit ratio only increases by 8%. In both replacement policies as the value of α increases and reaches 1.8 and 2 the increase in the hit ratio by increasing the cache size decreases. As we can see in the LFU replacement policy when the cache size increases from 15 to 20 and the value of α = 2 the hit ratio does not increase.

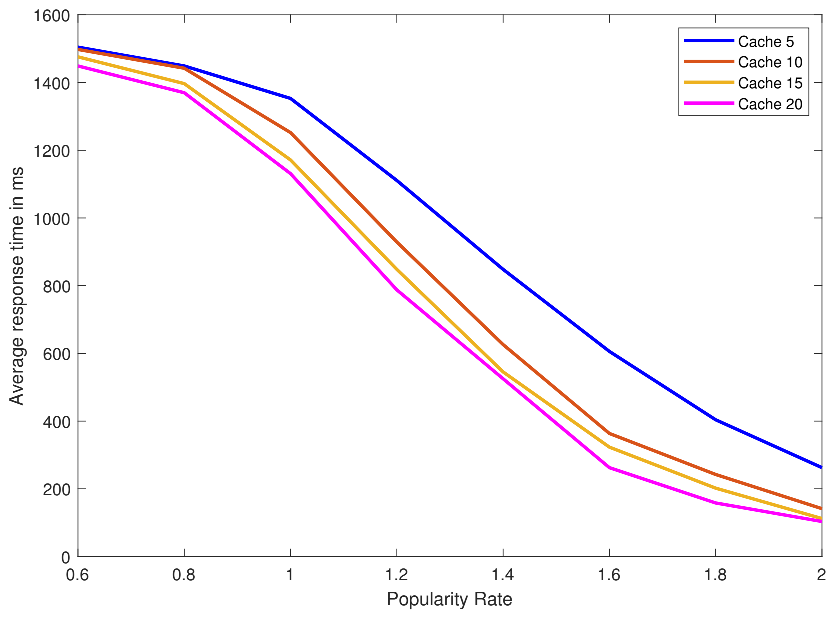

Figure 2. Average response time in ms for caches five, 10, 15 and 20 and different popularity rates using LRU cache replacement strategy and 2 Mbps link speed.

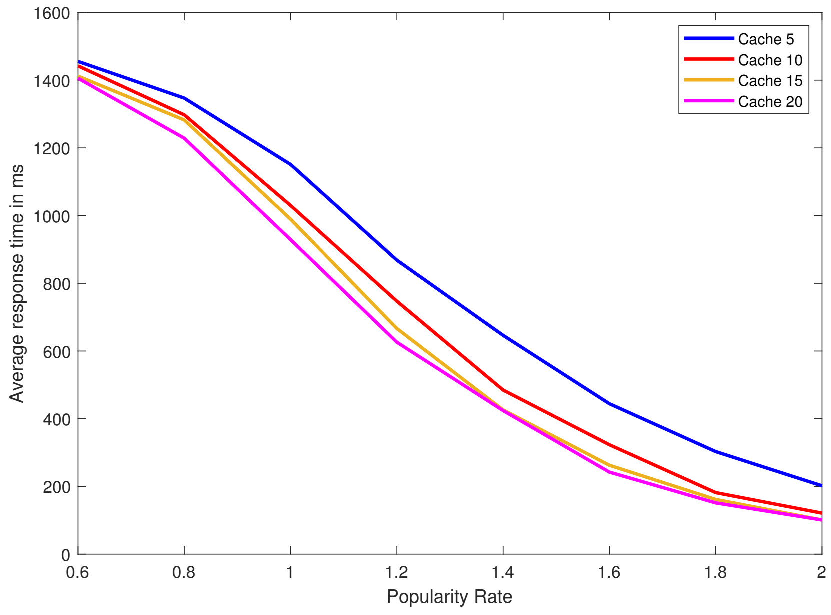

Figure 3. Average response time in ms for caches five, 10, 15 and 20 and different popularity rates using LFU cache replacement strategy.

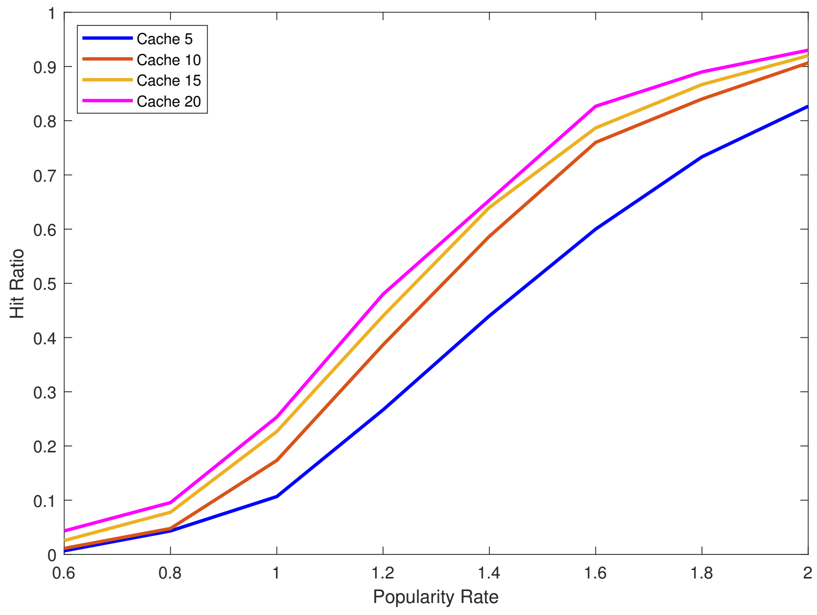

Figure 4. Hit ratio for caches five, 10, 15 and 20 and different popularity rates using LRU cache replacement strategy.

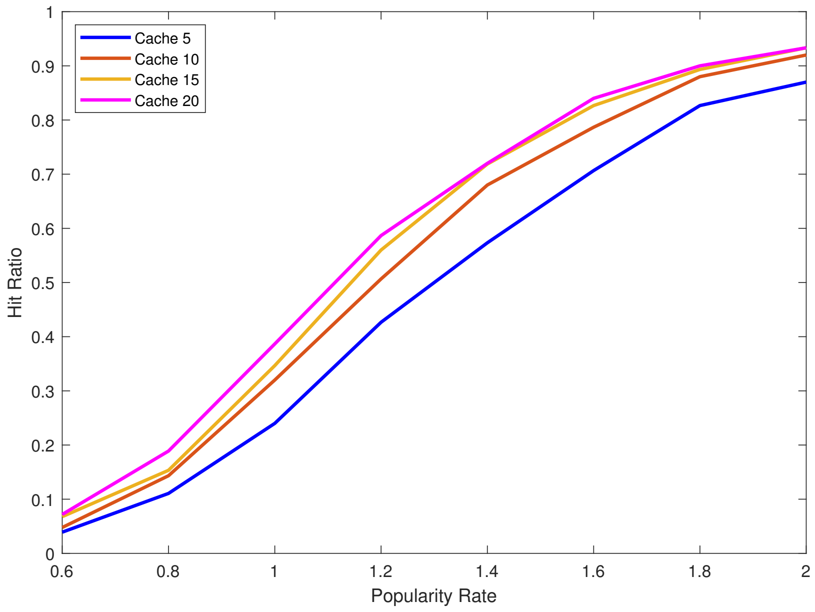

Figure 5. Hit ratio for caches five, 10, 15 and 20 and different popularity rates using LFU cache replacement strategy and 2 Mbps link speed.

6. Mathematical Assessment

The fundamental question here is how to approximate in a closed form solution the harmonic number $H_N(\alpha)$ given by Formula (2). For $\alpha = 1$, $H_N(1)$ is readily given by expression 0.131 in [54]; that is:

$$H_N(1) = \sum_{i=1}^{N} \frac{1}{i} \approx \gamma + ln(N) + \frac{1}{2N}$$

where γ is the Euler's constant and is given by $\gamma = 0.5772156649$. As the total number of data blocks, N, is suppose to be very large, we consider the following approximation instead:

$$H_N(1) \approx \gamma + ln(N) \tag{3}$$

and consequently we obtain:

$$P[X = r, 1] \approx \frac{r^{-\alpha}}{\gamma + ln(N)} \qquad \alpha = 1 \tag{4}$$

For $0 < \alpha < 1$, the integral approximation provides a rather accurate and useful closed form approximation of $H_N(\alpha)$.

$$H_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha} = \int_1^{N+1} x^{-\alpha} dx = \frac{(n+1)^{1-\alpha} - 1}{1-\alpha} \tag{5}$$

and consequently we get:

$$P[X = r, \alpha] \approx \frac{(1-\alpha) r^{-\alpha}}{(n+1)^{1-\alpha} - 1} \qquad 0 < \alpha < 1 \tag{6}$$

Formula (6) has been very successfully used in several work on caching that assumed a Zipf parameter (a popularity index) less than 1. However Formula (6) cannot be used for a Zipf parameter larger than 1 as it yields an unacceptable large relative error. The relative error of the use of Formula (6) instead of the original Formula (1) grows very quickly with α and attains more than 60% for $\alpha = 2$ [55].

In [55], the author noticed that approximate Formula (5) is always smaller than the actual sum of Formula (2), as it is an integral from below. Consequently, he proposed to approximate Formula (2) by taking the average of the integrals from above and from below, which yields:

$$H_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha} \approx \frac{(N+1)^{1-\alpha} + N^{1-\alpha} - (1+\alpha)}{2(1-\alpha)} \tag{7}$$

which once replaced in Formula (1) yields the so-called average integral approximation:

$$P_{ave}[X = r, \alpha] \approx \frac{2(1-\alpha)r^{-\alpha}}{(N+1)^{1-\alpha} + N^{1-\alpha} - (1+\alpha)} \tag{8}$$

As N, the total number of data blocks is very large, we may tacitly simplify Formula (7) to give:

$$H_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha} \approx \frac{(2N)^{1-\alpha} - (1+\alpha)}{2(1-\alpha)} \qquad \alpha \neq 1 \tag{9}$$

which provides a much better closed form expression for the truncated Zeta function for our considered range of α ($0 < \alpha \leq 2$ and $\alpha \neq 1$) given by:

$$P_{ave}[X = r, \alpha] \approx \frac{2(1-\alpha)r^{-\alpha}}{2N^{1-\alpha} - (1+\alpha)} \qquad \alpha \neq 1 \tag{10}$$

In fact, the relative error τ is given by:

$$\tau = \frac{P_{ave}[X = r, \alpha] - P[X = r, \alpha]}{P[X = r, \alpha]}$$

which yields for the case $\alpha = 1$ by using Formulas (1) and (4):

$$\tau = \frac{\sum_{i=1}^{N} \frac{1}{i}}{\gamma + lnN} - 1 \qquad \alpha = 1 \tag{11}$$

and yield for the case of $\alpha \neq 1$ by using Formulas (1) and (10):

$$\tau = \frac{2(1-\alpha)\sum_{i=1}^{N} i^{-\alpha}}{2N^{1-\alpha} - (1+\alpha)} - 1 \qquad \alpha \neq 1 \tag{12}$$

Table 6 provides the relative errors for the considered values of α and for N = 2048, 10,240 and 20,480. We clearly notice that our approximation of the truncated Zeta function given by Formula (10) is excellent when α is less than 1. For $\alpha = 1$, the approximation given by Formula (4) is also excellent. Our approximation as per Formula (10) for $\alpha > 1$, is still a good approximation but not as excellent as in the cases using lower values of α. We notice that even for a very high value of $\alpha = 2$, the relative error is still under 10%. We also notice that the relative error is not that sensitive to the value of N especially for large values of α; N = 2048 seems to be large enough.

Table 6. Error rate for different popularity rates and different working set size.

Popularity Rate	*N* = 2048	*N* = 10,240	*N* = 20,480
0.2	0.000224	0.000047	0.000024
0.4	0.000346	0.000104	0.000064
0.6	0.001033	0.000500	0.000372
0.8	0.003441	0.002307	0.001962
1	0.000029	0.000004	0.000002
1.2	0.020770	0.019440	0.019027
1.4	0.036632	0.035927	0.035744
1.6	0.055409	0.055135	0.055078
1.8	0.075681	0.075593	0.075572
2	0.096654	0.096629	0.096625

6.1. Cache Size and Caching Hit Ratio

Let $\psi[k,\alpha]$ be the cumulative probability of accessing the k most frequent (i.e., popular) data blocks. Then:

$$\psi[k,\alpha] = \sum_{r=1}^{k} P[X=r,\alpha] = \sum_{r=1}^{k} \frac{r^{-\alpha}}{\sum_{i=1}^{N} i^{-\alpha}} = \frac{H_k(\alpha)}{H_N(\alpha)} \tag{13}$$

Now using our approximations of the harmonic number of order N and α, we obtain for the case $\alpha = 1$ by using Formula (3):

$$\psi[k,1] \approx \frac{\gamma + lnk}{\gamma + lnN} \qquad \alpha = 1 \tag{14}$$

and for the case $\alpha \neq 1$ that is using Formula (9):

$$\psi[k,\alpha] \approx \frac{2k^{1-\alpha} - (1+\alpha)}{2N^{1-\alpha} - (1+\alpha)} \qquad \alpha \neq 1 \tag{15}$$

Let N_{req} denotes the total number of user requests; N_{req} is supposedly very large. The number of accesses made to the k most frequent data blocks, denoted by $N_{req,k}$, is then given by:

$$N_{req,k} = \sum_{r=1}^{k} N_{req} P[X=r,\alpha] \tag{16}$$

that is:

$$N_{req,k} = \sum_{r=1}^{k} N_{req} \frac{r^{-\alpha}}{\sum_{i=1}^{N} i^{-\alpha}} = N_{req} \frac{H_k(\alpha)}{H_N(\alpha)} \tag{17}$$

and by using Formula (13), we obtain:

$$N_{req,k} = N_{req}\psi[k,\alpha] \tag{18}$$

Now, if the k most frequent data blocks are always cached independently from the specifics of any used caching scheme, then the cache hit ratio C_{hit} is given by:

$$C_{hit} = \frac{N_{req,k}}{N_{req}} = \psi[k,\alpha]$$

which yields for the case $\alpha = 1$ by using Formula (14):

$$C_{hit} \approx \frac{\gamma + lnk}{\gamma + lnN} \qquad \alpha = 1 \tag{19}$$

and for the general case of $\alpha \neq 1$ by using approximation (15):

$$C_{hit} \approx \frac{2k^{1-\alpha} - (1+\alpha)}{2N^{1-\alpha} - (1+\alpha)} \qquad \alpha \neq 1 \tag{20}$$

Given k, this is the best cache hit ration that can be accomplished by any caching scheme subject to the relative error of our approximations. Finally (20) yields the following approximations of the cache size, denoted by C_{size}, for a given cache hit ratio C_{hit}:

$$C_{size} \approx e^{C_{hit}(lnN+\gamma)-\alpha} \qquad \alpha = 1 \tag{21}$$

$$C_{size} \approx \left[\frac{(2N^{1-\alpha} - (1+\alpha))c_{hit} + (1+\alpha)}{2}\right]^{\frac{1}{1-\alpha}} \qquad \alpha \neq 1 \tag{22}$$

6.2. Validation of the Mathematical Model

To validate our mathematical model, we used our experimental results. Figures 6 and 7 compare the theoretical and experimental hit ratios for cache sizes five and 20 when using LFU and LRU respectively. We clearly observe on Figure 6 the close match between the theoretical results given by Formulas (19) and (20) and the experimental results using the replacement strategy LFU. On Figure 7, we remarkably notice that both theoretical and experimental curves have the exact shape (behavior) but with almost 10% difference. This is essentially due to the use of the replacement strategy LRU that cannot accomplish as much as the theoretical case which provides the optimal (within the relative error) hit ratio. Recall that the theoretical model assumes that the k most frequent data blocks are always cached, and consequently it tacitly provides an upper bound on the value of the hit ratio. Figure 6 clearly shows that LFU leads to a hit ratio very close to this upper bound. Figure 7 shows, however, that LRU is not as efficient as LFU. To sum up, the experimental results are in complete synergy with our theoretical results and the mathematical model is very well validated.

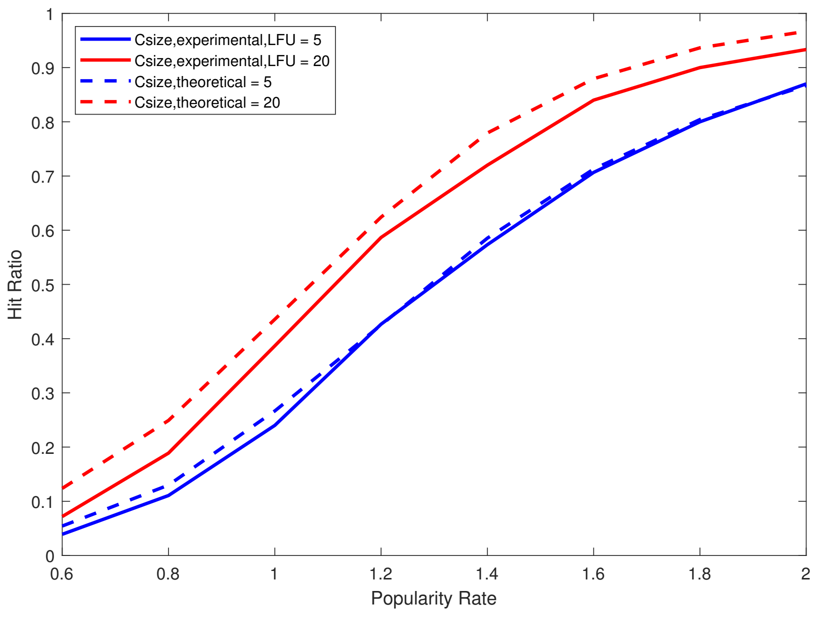

Figure 6. A comparison between the theoretical hit ratio and the experimental hit ratio using LFU for cache sizes five and 20 and different popularity rates.

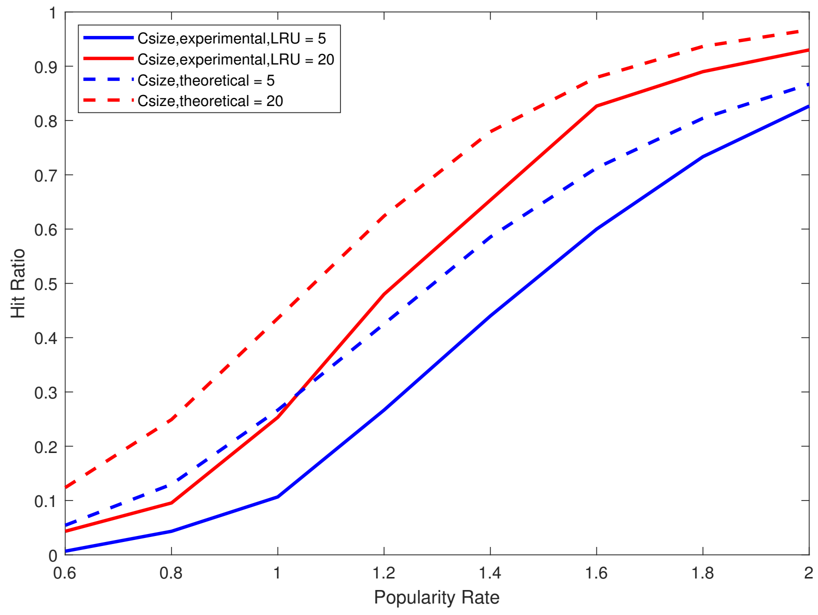

Figure 7. A comparison between the theoretical hit ratio and the experimental hit ratio using LRU for cache sizes five and 20 and different popularity rates and a working set size N = 2048.

Finally, Figure 8 portrays the required cache size for the different considered values of the popularity parameter α and for a hit ration of 60%. First of all, we observe that for values of α larger than one, we only need a very small cache. However, for a small value of α, the required cache size is much bigger. Notice also that for large popularity rates, the cache size becomes much less sensitive to the total number N of real blocks.

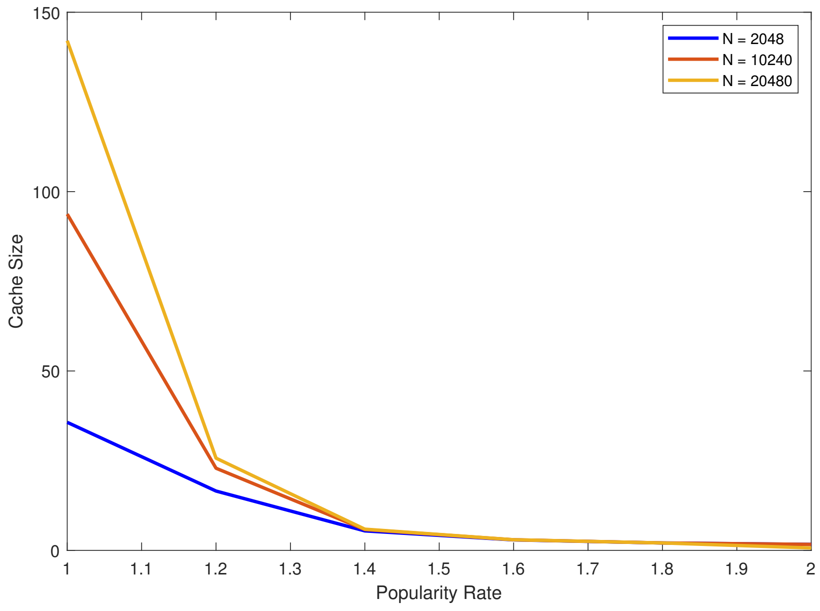

Figure 8. Cache size needed to achieve 60% hit ratio for different popularity rates.

7. Conclusions

We proposed a simple enhanced implementation of Path ORAM that takes advantage of any existent popularity and locality in the sequence of requested blocks. This required the addition of a small extra storage on the client side managed as a cache. We set up an experimental test bed and experimented our proposal using two cache replacement strategies LRU and LFU. The experimental results clearly show the merit of our enhancement as it outperforms the original Path ORAM even when using a small cache size and a rather reduced popularity rate. The question arose as how to calibrate the cache size. We presented a mathematical model that provided very good closed form approximation of the cache size and the hit ratio. Last but not least, we validated our mathematical model using the obtained experimental results.

While we have experimented with Path ORAM, the enhancement remains valid and can be applied to any ORAM technique as it does not depend on the specifics of Path ORAM. The mathematical model for large values of the popularity parameter could be further enhanced.

Author Contributions: Conceptualization, K.A.-S. and A.B.; Methodology, K.A.-S. and A.B.; Software, K.A.-S.; Validation, K.A.-S. and A.B.; Formal analysis, K.A.-S. and A.B.; Investigation, K.A.-S. and A.B.; Resources, K.A.-S. and A.B.; Data curation, K.A.-S.; Writing—original draft preparation, K.A.-S.; Writing—review and editing, K.A.-S. and A.B.; Visualization, K.A.-S. and A.B.; Supervision, A.B.; Project administration, A.B.; Funding acquisition, K.A.-S.

Funding: King Abdulaziz City for Science and Technology (KACST) for funding this research with research grant number (1-17-00-001-0004).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gao, C.Z.; Cheng, Q.; Li, X.; Xia, S.B. Cloud-assisted privacy-preserving profile-matching scheme under multiple keys in mobile social network. *Cluster Comput.* **2018**, 1–9. [CrossRef]
2. Islam, M.S.; Kuzu, M.; Kantarcioglu, M. *Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation*; NDSS: New York, NY, USA, 2012; Volume 20, p. 12.
3. Ostrovsky, R. Efficient Computation on Oblivious RAMs. In Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, 13–17 May 1990; ACM: New York, NY, USA, 1990; pp. 514–523. [CrossRef]
4. Goldreich, O. Towards a theory of software protection and simulation by oblivious RAMs. In Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 25–27 May 1987; pp. 182–194. [CrossRef]
5. Goldreich, O.; Ostrovsky, R. Software protection and simulation on oblivious RAMs. *J. ACM* **1996**, *43*, 431–473. [CrossRef]
6. Pinkas, B.; Reinman, T. Oblivious RAM Revisited. In *Advances in Cryptology—CRYPTO 2010*; Rabin, T., Ed.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 502–519. [CrossRef]
7. Goodrich, M.T.; Mitzenmacher, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*; Springer: Berlin, Germany, 2011; pp. 576–587. [CrossRef]
8. Goodrich, M.T.; Mitzenmacher, M.; Ohrimenko, O.; Tamassia, R. Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation. In Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, Kyoto, Japan, 17–19 January 2012; pp. 157–167. [CrossRef]
9. Shi, E.; Chan, T.H.H.; Stefanov, E.; Li, M. Oblivious RAM with O((logN)3) Worst-Case Cost. In *Advances in Cryptology—ASIACRYPT 2011*; Lee, D.H., Wang, X., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 197–214.
10. Kushilevitz, E.; Lu, S.; Ostrovsky, R. On the (in)Security of Hash-based Oblivious RAM and a New Balancing Scheme. In Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, Kyoto, Japan, 17–19 January 2012; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2012; pp. 143–156.
11. Stefanov, E.; Shi, E.; Song, D. Towards Practical Oblivious RAM. *arXiv preprint* **2011**, arXiv:1106.3652.

12. Stefanov, E.; van Dijk, M.; Shi, E.; Fletcher, C.; Ren, L.; Yu, X.; Devadas, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; ACM: New York, NY, USA, 2013; pp. 299–310. [CrossRef]
13. Wang, X.S.; Huang, Y.; Chan, T.H.H.; Shelat, A.; Shi, E. SCORAM: Oblivious RAM for Secure Computation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; ACM: New York, NY, USA, 2014; pp. 191–202. [CrossRef]
14. Liu, C.; Wang, X.S.; Nayak, K.; Huang, Y.; Shi, E. Oblivm: A programming framework for secure computation. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 359–376. [CrossRef]
15. Ren, L.; Fletcher, C.; Kwon, A.; Stefanov, E.; Shi, E.; van Dijk, M.; Devadas, S. Constants Count: Practical Improvements to Oblivious RAM. In Proceedings of the 24th USENIX Security Symposium (USENIX Security '15), Washington, DC, USA, 12–14 August 2015; pp. 415–430.
16. Devadas, S.; van Dijk, M.; Fletcher, C.W.; Ren, L.; Shi, E.; Wichs, D. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography: Proceedings of the 13th International Conference, TCC 2016-A, Tel Aviv, Israel, 10–13 January 2016, Part II*; Springer: Berlin, Germany, 2016; pp. 145–174.
17. Xie, D.; Li, G.; Yao, B.; Wei, X.; Xiao, X.; Gao, Y.; Guo, M. Practical private shortest path computation based on oblivious storage. In Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering (ICDE), Helsinki, Finland, 16–20 May 2016; pp. 361–372. [CrossRef]
18. Asharov, G.; Chan, T.H.H.; Nayak, K.; Pass, R.; Ren, L.; Shi, E. Oblivious Computation with Data Locality. Cryptology ePrint Archive, Report 2017/772; 2017. Available online: https://eprint.iacr.org/2017/772 (accessed on 12 May 2018).
19. Stefanov, E.; Dijk, M.V.; Shi, E.; Chan, T.H.H.; Fletcher, C.; Ren, L.; Yu, X.; Devadas, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* **2018**, *65*, 18:1–18:26. [CrossRef]
20. Ren, L.; Fletcher, C.W.; Kwon, A.; Stefanov, E.; Shi, E.; van Dijk, M.; Devadas, S. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* **2014**, *2014*, 997.
21. Bao, C.; Srivastava, A. Exploring timing side-channel attacks on path-ORAMs. In Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 1–5 May 2017; pp. 68–73. [CrossRef]
22. Gao, W.; Iyengar, A.K.; Srivatsa, M. Caching Provenance Information. US Patent 8,577,993, 11 May 2013.
23. Pirk, H.; Grund, M.; Krueger, J.; Leser, U.; Zeier, A. *Cache Conscious Data Layouting for in-Memory Databases*; Hasso-Plattner-Institute: Potsdam, Germany, 2010.
24. Chang, Z.H. Real-time distributed video transcoding on a heterogeneous computing platform. In Proceedings of the 2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Jeju, Korea, 25–28 October 2016.
25. Ubarhande, V. Performance Based Data-Distribution Methodology In Heterogeneous Hadoop Environment. Ph.D. Thesis, National College of Ireland, Dublin, Ireland, 2014.
26. Chang, Z.; Xie, D.; Li, F. Oblivious RAM: A Dissection and Experimental Evaluation. *Proc. VLDB Endow.* **2016**, *9*, 1113–1124. [CrossRef]
27. Teeuwen, P. Evolution of Oblivious RAM Schemes. Master's Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2015.
28. Fletcher, C.W. Oblivious RAM: From Theory to Practice. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2016.
29. Ren, L.; Yu, X.; Fletcher, C.W.; van Dijk, M.; Devadas, S. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. *SIGARCH Comput. Archit. News* **2013**, *41*, 571–582. [CrossRef]
30. Fletcher, C.W.; Ren, L.; Yu, X.; Van Dijk, M.; Khan, O.; Devadas, S. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In Proceedings of the 2014 IEEE 20th International Symposium onHigh Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 213–224.
31. Maas, M.; Love, E.; Stefanov, E.; Tiwari, M.; Shi, E.; Asanovic, K.; Kubiatowicz, J.; Song, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; ACM: New York, NY, USA, 2013; pp. 311–324. [CrossRef]

32. Yu, X.; Haider, S.K.; Ren, L.; Fletcher, C.; Kwon, A.; van Dijk, M.; Devadas, S. PrORAM: Dynamic Prefetcher for Oblivious RAM. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; ACM: New York, NY, USA, 2015; pp. 616–628. [CrossRef]
33. Ren, L.; Fletcher, C.; Yu, X.; Kwon, A.; van Dijk, M.; Devadas, S. Unified Oblivious-RAM: Improving Recursive ORAM with Locality and Pseudorandomness. Cryptology ePrint Archive, Report 2014/205; 2014. Available online: https://eprint.iacr.org/2014/205 (accessed on 12 May 2018).
34. Fletcher, C.W.; Ren, L.; Kwon, A.; van Dijk, M.; Devadas, S. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, Turkey, 14–18 March 2015; ACM: New York, NY, USA, 2015; pp. 103–116. [CrossRef]
35. Zhang, X.; Sun, G.; Zhang, C.; Zhang, W.; Liang, Y.; Wang, T.; Chen, Y.; Di, J. Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses. In Proceedings of the 48th International Symposium on Microarchitecture, Waikiki, HI, USA, 5–9 December 2015; ACM: New York, NY, USA, 2015; pp. 102–114. [CrossRef]
36. Sánchez-Artigas, M. Enhancing Tree-Based ORAM Using Batched Request Reordering. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 590–604. [CrossRef]
37. Chakraborti, A.; Sion, R. SeqORAM: A Locality-Preserving Write-Only Oblivious RAM. *arXiv Preprint* **2017**, arXiv:1707.01211.
38. Li, L.; Datta, A. Write-only oblivious RAM-based privacy-preserved access of outsourced data. *Int. J. Inf. Secur.* **2017**, *16*, 23–42. [CrossRef]
39. Wong, K.Y. Web cache replacement policies: A pragmatic approach. *IEEE Netw.* **2006**, *20*, 28–34. [CrossRef]
40. Zotano, M.G.; Gómez-Sanz, J.; Pavón, J. Impact of traffic distribution on web cache performance. *Int. J. Web Eng. Technol.* **2015**, *10*, 202–213. [CrossRef]
41. Lee, D.; Choi, J.; Kim, J.H.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* **2001**, *50*, 1352–1361. [CrossRef]
42. Ronao, C.A.; Cho, S.B. Anomalous query access detection in RBAC-administered databases with random forest and PCA. *Inf. Sci.* **2016**, *369*, 238–250. [CrossRef]
43. Takatsuka, Y.; Nagao, H.; Yaguchi, T.; Hanai, M.; Shudo, K. A caching mechanism based on data freshness. In Proceedings of the 2016 International Conference on Big Data and Smart Computing (BigComp), Hong Kong, China, 18–20 January 2016; pp. 329–332. [CrossRef]
44. Kamra, A.; Terzi, E.; Bertino, E. Detecting Anomalous Access Patterns in Relational Databases. *VLDB J.* **2008**, *17*, 1063–1077. [CrossRef]
45. Hasslinger, G.; Ntougias, K. Evaluation of Caching Strategies Based on Access Statistics of Past Requests. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*; Fischbach, K., Krieger, U.R., Eds.; Springer International Publishing: Cham, Switzerland, 2014; pp. 120–135.
46. Zipf, G.K. *Selected Studies of the Principle of Relative Frequency in Language*; Harvard University Press: Cambridge, MA, USA, 2014. [CrossRef]
47. Zipf, G.K. *Human Behavior and the Principle of Least Effort*; Addision-Wesley: Boston, MA, USA, 1950.
48. Kriege, J.; Buchholz, P. PH and MAP Fitting with Aggregated Traffic Traces. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*; Fischbach, K., Krieger, U.R., Eds.; Springer International Publishing: Cham, Switzerland, 2014; pp. 1–15.
49. Breslau, L.; Cao, P.; Fan, L.; Phillips, G.; Shenker, S. Web caching and Zipf-like distributions: Evidence and implications. In Proceedings of the IEEE INFOCOM ’99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320), New York, NY, USA, 21–25 March 1999; Volume 1, pp. 126–134. [CrossRef]
50. Shi, L.; Gu, Z.; Wei, L.; Shi, Y. Quantitative Analysis of Zipf’s Law on Web Cache. In *Parallel and Distributed Processing and Applications*; Pan, Y., Chen, D., Guo, M., Cao, J., Dongarra, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 845–852.
51. Nair, T.R.G.; Jayarekha, P. A Rank Based Replacement Policy for Multimedia Server Cache Using Zipf-Like Law. *arXiv Preprint* **2010**, arXiv:1003.4062.

52. Zotano, M. Analysis of web objects distribution. In Proceedings of the 12th International Conference on Distributed Computing and Artificial Intelligence, Salamanca, Spain, 3–5 June 2015; Volume 373, pp. 105–112. [CrossRef]
53. Tomé, C.G.; Garcıa, C.J. Adaptive Fragment Designs to Reduce the Latency of Web Caching in Content Aggregation Systems. Ph.D. Thesis, Universitat de les Illes Balears, Palma, Spain, 2017.
54. Gradshteyn, I.; Ryzhik, I. *Table of Integrals, Series, and Products*; Academic Press: New York, NY, USA, 1980.
55. Naldi, M. Approximation of the truncated Zeta distribution and Zipf's law. *arXiv preprint* **2015**, arXiv:1511.01480.