UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Effectiveness of Oblivious RAM in cloud storage services

Jacco Brandt
M.Sc. Thesis
August, 2022

Supervisors:
dr.ing. F.W. Hahn
dr. R. Holz

External Committee Member:

Services and CyberSecurity
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Effectiveness of ORAM in cloud storage services

Jacco Brandt
s1423290
University of Twente
CyberSecurity (4TU)
j.h.brandt@student.utwente.nl

ABSTRACT

Cloud storage services provide their users with external storage space that can be used to save files on an external server. Such storage services usually support encryption techniques because of privacy and security considerations. Regular encryption techniques can guarantee that the content of these files remains confidential, however these methods cannot hide the access patterns of the files. Oblivious RAM(ORAM) algorithms are techniques that are used to protect both the content and access pattern of stored data. In this paper we measure how to transform these files into blocks suitable for usage by ORAM algorithms efficiently, and compare three of the most well-known ORAM algorithms to verify which are best suited for usage in cloud storage services, namely Square-root ORAM, trivial ORAM and PathORAM. Our results show that both Square-root ORAM and PathORAM are suitable for usage in cloud storage services, but their usage scenarios differ. Square-root ORAM is best suited for performing automated backups, because it has a high worst-case performance which regularly causes requests to take a long time. PathORAM is best suited in situations where cloud storage services are used to reduce the local storage by storing files externally, because it has a constant cost performance ensuring stable and fast reponse to requests.

1. INTRODUCTION

In the cloud storage services industry the providers usually allow the users to upload plaintext or regularly encrypted files to their storage. However, regular encryption does not provide a complete privacy guarantee [3]. While this method may secure the content of the files, it does not hide the access patterns. This means that the server (or malicious third party) can observe which files are accessed at which frequency, and use this information for potential malicious purposes as is shown by Islam et al. [11]. A potential way to protect this information is that the cloud storage provider should provide ORAM storage in which the client can securely store data [18].

ORAM storage is essentially a group of algorithms that store encrypted information in a special way, so that queries do not leak any information about the access patterns. There are many different implementations of ORAM each with their own specific properties, such as Square-root ORAM [5], Trivial ORAM [20] and PathORAM [22]. Some implementations require large amounts of client-side storage, while others can securely store the client-side state in the ORAM storage so that multiple clients can be used to access it. There are also large differences in the efficiency of storage and in communication complexity between various implementations, which affect the usability of the implementation in specific use-cases.

1.1. Problem statement and research goal

Regular cloud storage services usually store their files in normal plaintext format, or the server encrypts the entire file and stores it as such. In this case the encryption key is manager by the server. The advantage of this is that the client does not need to manage the key and can access the cloud storage from any device, while ensuring that if the stored files are leaked or hacked the content is still protected. The disadvantage is that the cloud storage provider has access to the encryption key and has the ability to decrypt the files that are stored. A more advanced approach of encryption is the situation in which the client itself manages the encryption key, and encrypts the files before uploading them to the cloud storage server. This ensures that even the server itself cannot access the plaintext files. However, the server (and other malicious observers) can still observe access patterns and obtain potential privacy-sensitive information about the files stored on the server, even if they are encrypted. For example, the approximate size of the files can be judged based on the encrypted file size, the frequency with which files are accessed is easily observed, and the differences between read and write operations are also obvious.

In order to make up for the shortcomings of the use of regular encryption in cloud storage services, we use ORAM algorithms to store files on the server instead of regular encryption. The goal of this research is to find out which ORAM solutions are best suited for cloud storage services and how to properly implement it.

Research question: What ORAM solutions are best-suited for cloud storage services?

To answer this research question, we need to analyse the answers of the 3 research sub-questions that are introduced in section 4 methodology. This section describes how we can store files in a data format used by ORAM algorithms, as well as how to measure and analyse the efficiency and other properties of these algorithms.

1.2. Our contribution

In order to find an answer to the previously posed research question, we created python code that implements a client with a local storage directory and a server representing the cloud storage service, as is further described in section 4. The client consists of code that maps the local storage directory into content(blocks) that can be used by ORAM algorithms. We measure various methods in which we to transform files into blocks and analyse which is most suitable for cloud storage services, as described in section 5. By writing and reading these blocks to the server using different ORAM algorithms with various options, we measure which are best suited for cloud storage purposes as described in section 6. In section 7 we combine our results in order to address the research question and argue that both PathORAM and Square-root ORAM are well-suited for cloud storage services. However, the scenarios for which they are suitable are different. PathORAM is best used in cloud storage services where the client does not keep a local copy of the stored files, such as mobile phones or other scenarios in which the user wishes to reduce local storage usage. While Square-root ORAM is best suited for cloud storage services used as back-up, where the local directory can be synchronised with the cloud storage service in a background process.

The code that implements the file conversion method, the ORAM implementations, as well as the measurement scripts and results are all available on git repository the: `https://gitlab.utwente.nl/s1423290/oram-cloud-storage-service/`

2. BACKGROUND

2.1. Cloud storage services

Local storage media, while efficient, are often limited in size and easily susceptible to data loss [29]. In order to provide users and businesses with larger storage space and data safety, there are many companies that offer cloud storage services such as Dropbox and iCloud. These services are specialised in providing access to dedicated storage space that can be accessed over the internet. These services generally provide guarantees that are difficult or expensive to ensure for local storage media, such as data integrity, data safety (backup systems) and availability (crashes will not cause a loss of availability) [29]. Besides data integrity and safety, the cloud service providers also have a duty to ensure that the data stored is secure. Besides limiting user access through detailed access management models, there is also the need to encrypt the stored data so that a potential leakage will not compromise the confidentiality of the data [13]. There are many different methods to do so, each of them have different advantages and disadvantages. The list below gives an overview of several promising solutions.

- Regular (a)symmetric encryption can be used to encrypt the files before transmitting them to the cloud storage service. This is fast and efficient, but has the drawback that, besides the file contents, all other information (such as access patterns, filename and other meta-data) is leaked to the cloud storage service.
- (A)Symmetric Searchable Encryption (ASE/SSE) are techniques used to store the encrypted data on the server under so-called encrypted tokens [12, Section 4.1]. This technique enables the user to search through the encrypted content in the database, even though the cloud storage service has no knowledge about the data except for the encrypted content. However, by analysing the access pattern the (honest-but-curious) cloud storage service can learn which encrypted tokens refer to which encrypted file, which files are accessed at what times and when the content of files are updated.
- (Fully) Homomorphic Encryptions are encryption techniques that can be used to perform calculations/operations over encrypted data to calculate the ciphertext of the calculation result [19]. Such a method can be very useful in allowing the server to perform calculations over stored ciphertext, essentially performing cloud computing without sharing the decryption keys with the server [23]. While this method ensures that the plaintext content of the stored data and calculation results are protected, the server can still analyse the calculation steps that are performed and analyse which content is used in the calculation.
- Private Information Retrieval (PIR) are techniques used to retrieve items from a database while masking which item has been retrieved [2]. Most methods only work with distributed servers that have several nodes, making it easier to mask which specific item the client requires. The single database solutions often work by requesting a group of items from the database, one of which is the item that is required by the client. This ensures that the server does not know which specific item is required by the client, but it can analyse multiple requests to narrow down the information over time.

Even though the above mentioned techniques all provide different methods to protect the data stored in the cloud service provider, each method has its own weaknesses as well. Private Information Retrieval (PIR) may protect which specific database entry has been requested, yet it provides no protection for the (plaintext) content stored. Even though the 3 encryption methods (regular, searchable and homomorphic) protect the content stored on the server, the server will still be able to analyse the access pattern. If the storage server has been compromised by an adversary who observes the access pattern, this information can still be used to facilitate cyber-attacks and/or theft of information. Even if the storage server has not been compromised by a third party, there is no guarantee that the server does not (un)intentionally leak such information. In order to protect both the content and the access pattern from being analysed by the server and/or a third party, there is a suitable (group of) algorithms that can be applied. Oblivious RAM (ORAM) can store encrypted information in a way that access queries do not leak any information about the access patterns, and will be described in more detail in subsection 2.2.

2.2. Oblivious RAM

Oblivious RAM (ORAM) algorithms were first introduced by Goldreich and Ostrovsky in order to protect software from piracy by concealing the content and access pattern of program instructions during execution [4, 5]. Because ORAM algorithms provide additional protection compared to regular encryption techniques it can also be used to store data on another machine, some research in this field focuses on improving the efficiency of the algorithms in a client-server scenario [28].

There have been many different ORAM algorithms designed over the years, all of which conform to the standard ORAM security definition. We adopt the standard ORAM security definition from [21]:
Standard definition: Intuitively, the security definition requires that the server learns nothing about the access pattern. In other words, no information should be leaked about:

1. which data is being accessed;
2. how old it is (when it was last accessed);
3. whether the same data is being accessed (linkability);
4. access pattern (sequential, random, etc);
5. whether the access is a read or a write.

Definition 1: (Formal security definition). Let $\vec{y} := ((op_1, a_1, data_1), (op_2, a_2, data_2), ..., (op_M, a_M, data_M))$ denote a data request sequence of length M, where each op_i denotes a read(u_i) or a write(u_i, data) operation. Specifically, u_i denotes the identifier of the block being read or written, and $data_i$ denotes the data being written. Let $A(\vec{y})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\vec{y}$. An ORAM construction is said to be secure if for any two data request sequences $\vec{y}$ and $\vec{z}$ of the same length, their access patterns $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable by anyone but the client.

2.2.1. Square-root ORAM

Square-root ORAM is one of the ORAM algorithms introduced by Goldreich and Ostrovsky to conceal the content and access patterns of program instructions [5, Section 4]. Even though the Square-root ORAM was originally meant to be used on a single machine, its usage in client-server scenarios can still protect access patterns.

Instead of a binary tree structure, which is commonly used by many ORAM algorithms, Square-root ORAM simply uses the ordinary (sequential) memory structure. Provided that the storage capacity of the server should be N blocks, then the server-side has to posses a memory of $N + \sqrt{N}$ blocks while the client-side has a shelter S of $\sqrt{N}$ blocks. The server-side memory contains N real blocks and $\sqrt{(N)}$ dummy blocks, while the client-side shelter starts out empty. The server-side memory (real & dummy blocks) will be obliviously permuted in a way that only the client-side knows which virtual memory address a corresponds to the real server-side memory address.

The Square-root ORAM algorithm works in epochs of exactly $\sqrt{N}$ data accesses. When accessing an address a (either read or write) the client-side will request its value from the server and store it in shelter S. If the client-side already has this address stored in the shelter, it will request a dummy-value from the server. During each epoch every dummy value can only be requested (at most) once in order to comply with the standard ORAM security definition. During the epoch all writes to the data will only be applied to the client-side shelter and not to the server-side storage. Only after an epoch has been completed will the client-side update the server-side memory, in a way so that the server-side memory has been obliviously permuted.

In order to use Square-root ORAM more efficiently in client-server scenarios, Zahur et al. suggests changes to the original algorithm [28]. Instead of using a hash-function to determine the position of each block as the original version by Goldreich does, Zahur et al. introduces a position map variable π that contains the position of each block. This position map can be easily obliviously permuted during initialisation and the end of every epoch, making these processes much more efficient then the previous oblivious sorting.

2.2.2. Trivial Bucket ORAM variant

Most ORAM schemes have very high worst-case performance costs($\Omega(N)$) because every epoch the whole structure must be re-organized/shuffled leading to a long delay, this property is very impractical in realistic situations [1]. The Trivial bucket ORAM variant introduced by Shi et al. [20, Section 3] introduces a new way to perform ORAM by spreading out the shuffle mechanism during every request, namely eviction. Eviction ensures that the worst-case performance will be reduced significantly ($O((logN)^3)$) and improves the practical applications of ORAM [20].

The data structure used by the server is a binary tree with N leaves and a depth of $D = log_2(N)$, each node of this tree is a bucket containing L blocks. Each of these buckets support the operations ReadAndRemove, Add and Pop.

Every new entry(block) into the ORAM structure will be added to the root bucket and assigned a random leaf as position l in the local clients' position map. Whenever a block must be read from the ORAM structure, the client will perform the ReadAndRemove operation on every bucket between its leaf bucket and the root bucket. Afterwards this block will be assigned a new position(leaf)

l^* and gets added to the root bucket using the Add operation. Both the ReadAndRemove and the Add operation are detailed in figure 1 displayed on the next page.

Eviction

In order to avoid the overflow of the buckets and also to avoid the high cost worst case performances, this scheme performs the eviction algorithm after every read or write operation [20, section 3.2]. In this algorithm the client will pop a certain amount v of buckets from every layer(depth) of the tree except for the leaves. The popped block will be added to the child bucket that matches its position, while a dummy value will be written to the other child node of the bucket. The details of the Evict operation are described in figure 2 displayed on the next page, while the process is illustrated by figure 3.

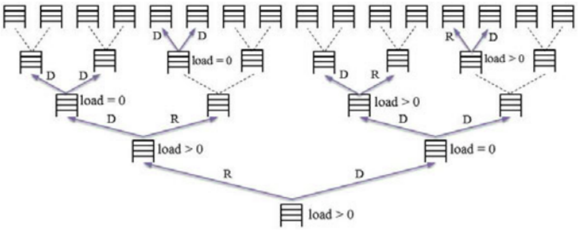


Figure 3: Eviction process [20]. *R* represents a real block. *D* represents a dummy block. *load* represents the amount of real blocks stored in a bucket

2.2.3. PathORAM

PathORAM is an ORAM algorithm proposed by Stefanov et al. that is efficient and easy to implement [22]. The server uses a binary tree as data-structure to store information, an overview of such a data-structure has been included in figure 4. Every node in this binary tree is a bucket, and every bucket contains Z blocks of size B that can store data. With a binary tree of height L the position of the buckets are referred to by describing the path between the root bucket and a leaf node x (which is a value between 0 and 2^L). So the path $P(x)$ refers to all buckets between the root node and node x, while the position $P(x,l)$ refers to the bucket at a specific height (level) in the previously mentioned path. The server supports simple read and write operations that either reads all Z blocks of a bucket, or writes all Z blocks to a bucket, any empty blocks will be filled with dummy data and all blocks are encrypted by the client before submission.

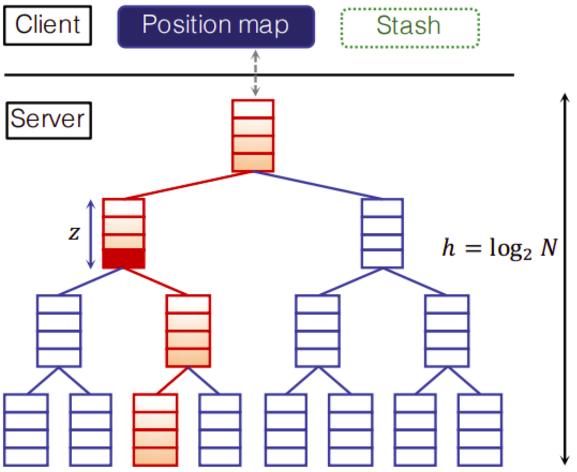


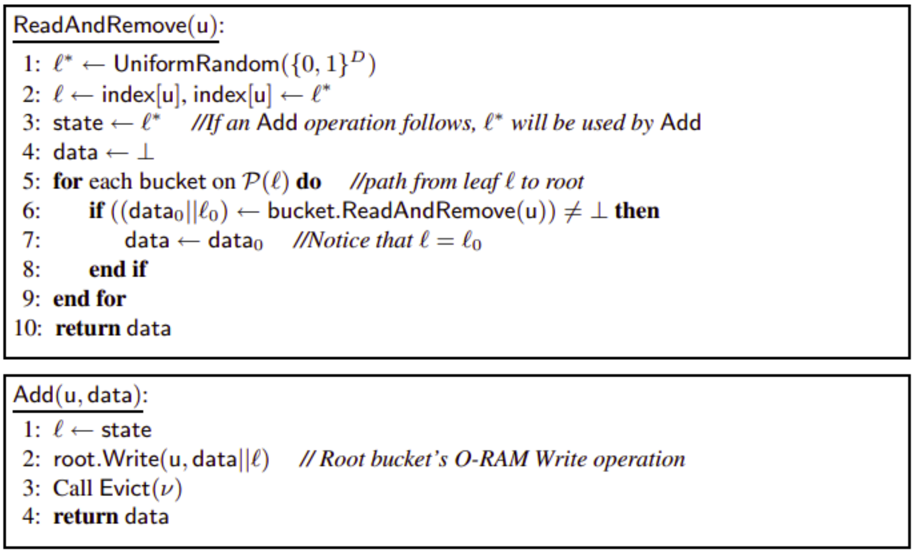
Figure 4: Binary tree data structure [9]

The client side only has to store 2 separate data structures. The first one is a temporary storage called a stash S, which temporarily stores any read blocks to save them in other buckets. The second local data structure will be the position map $position[a]$, this structure is used to find in which path $P(x)$ a block representing address a is stored. Without this data structure it becomes impossible to access the data stored in the binary tree on the server, thus this ORAM algorithm is only usable in a single client. If anyone wants to use this algorithm to service multiple clients they need to adjust the protocol so that the position map is stored online as well.

The PathORAM algorithm is based on the Trivial Bucket ORAM variant of shi et al., but it implements a different way to perform eviction during the access operations. Stefanov et al. describe the client-side protocol as depicted in figures 5 displayed on the next page. The notations used in this figure are described in table 1.

N	Total # blocks outsourced to server
L	Height of binary tree
B	Block size (in bits)
Z	Capacity of each bucket (in block)
$P(x)$	Path from leaf node x to the root
$P(x,l)$	The bucket at level l along the path $P(x)$
S	Client's local stash
position	Client's local position map
x := position[a]	Block a is currently associated with leaf node x, i.e., block a resides somewhere along $P(x)$ or in the stash.

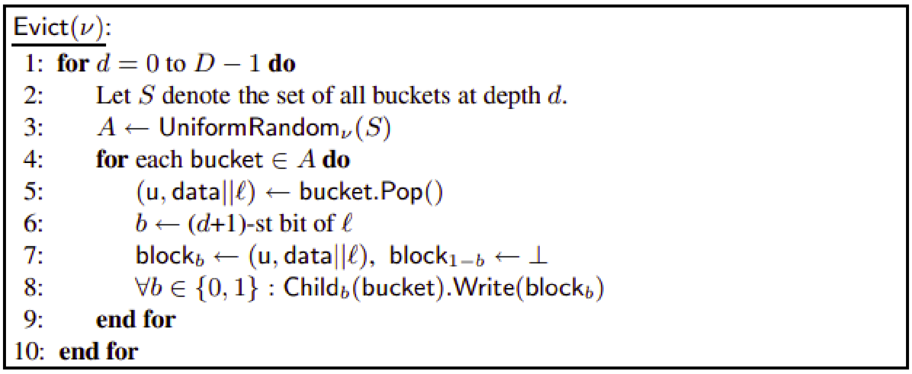
Table 1: PathORAM notations [22]



```
ReadAndRemove(u):
  $\ell^* \leftarrow$ UniformRandom($\{0,1\}^D$)
  $\ell \leftarrow$ index[u], index[u] $\leftarrow \ell^*$
  state $\leftarrow \ell^*$    //If an Add operation follows, $\ell^*$ will be used by Add
  data $\leftarrow \perp$
  for each bucket on $\mathcal{P}(\ell)$ do    //path from leaf $\ell$ to root
      if ((data$_0||\ell_0$) $\leftarrow$ bucket.ReadAndRemove(u)) $\neq \perp$ then
          data $\leftarrow$ data$_0$    //Notice that $\ell = \ell_0$
      end if
  end for
  return data
```

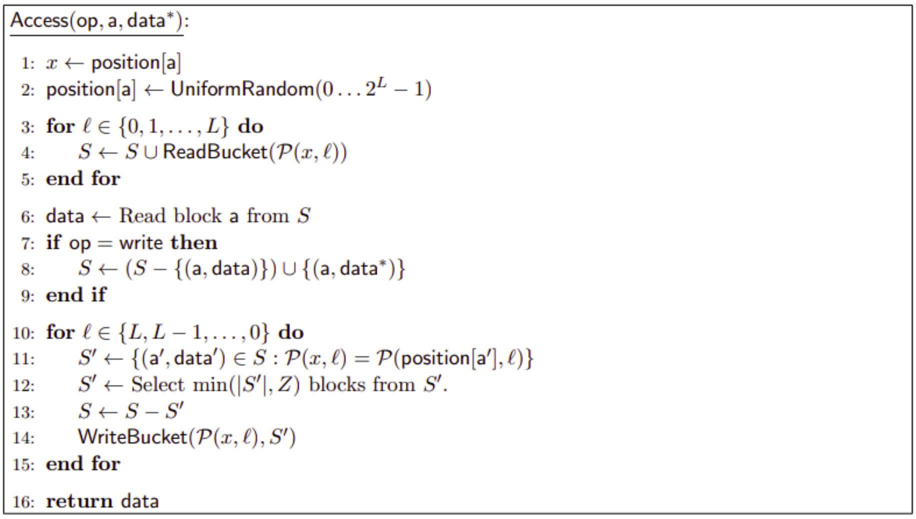
```
Add(u, data):
  $\ell \leftarrow$ state
  root.Write(u, data$||\ell$)    // Root bucket's O-RAM Write operation
  Call Evict($\nu$)
  return data
```

Figure 1: Data access algorithms [20]



```
Evict($\nu$):
  for $d = 0$ to $D - 1$ do
      Let $S$ denote the set of all buckets at depth $d$.
      $A \leftarrow$ UniformRandom$_\nu(S)$
      for each bucket $\in A$ do
          (u, data$||\ell$) $\leftarrow$ bucket.Pop()
          $b \leftarrow$ ($d$+1)-st bit of $\ell$
          block$_b \leftarrow$ (u, data$||\ell$), block$_{1-b} \leftarrow \perp$
          $\forall b \in \{0,1\}$ : Child$_b$(bucket).Write(block$_b$)
      end for
  end for
```

Figure 2: Eviction algorithm [20]



```
Access(op, a, data*):
  $x \leftarrow$ position[a]
  position[a] $\leftarrow$ UniformRandom($0 \ldots 2^L - 1$)

  for $\ell \in \{0, 1, \ldots, L\}$ do
      $S \leftarrow S \cup$ ReadBucket($\mathcal{P}(x, \ell)$)
  end for

  data $\leftarrow$ Read block a from $S$
  if op = write then
      $S \leftarrow (S - \{(a, data)\}) \cup \{(a, data^*)\}$
  end if

  for $\ell \in \{L, L-1, \ldots, 0\}$ do
      $S' \leftarrow \{(a', data') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(position[a'], \ell)\}$
      $S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$.
      $S \leftarrow S - S'$
      WriteBucket($\mathcal{P}(x, \ell), S'$)
  end for

  return data
```

Figure 5: PathORAM client-side instructions [22]

Meaning	**Notation**	**ORAM Scheme**
Total number of blocks (storage)	N	Square-root
Total number of leaf nodes/buckets	N	Trivial bucket
Total number of blocks (storage)	N	PathORAM
Number of blocks per bucket	L	Trivial bucket
Number of blocks per bucket	Z	PathORAM
Height/depth of tree	D	Trivial bucket
Height/depth of tree	L	PathORAM
Block address	a	Square-root
Block address	u	Trivial bucket
Block address	a	PathORAM
Position map	π	Square-root
Position map	$index[u]$	Trivial bucket
Position map	$position[a]$	PathORAM
Leaf position	l, l^*	Trivial bucket
Leaf position	x	PathORAM
Shelter	S	Square-root
Stash	S	PathORAM

Table 2: Notations used by various ORAM schemes

2.2.4. ORAM notation dictionary

The various ORAM papers all use different notations to refer to specific concepts, such as the tree height/depth, referred to as L in PathORAM and as D in the Trivial bucket ORAM variant. In our background description we used the same notations as the original sources used, so that the sources can be easily referenced while reading our background section. In order to illustrate the differences, this section contains a dictionary detailing the different notations used by the various ORAM schemes in table 2.

3. RELATED WORK

Oblivious RAM was first introduced by Goldreich and Ostrovsky for the purpose of software protection [4]. After its introduction most related research mainly focused on improving the communication complexity [1], [7], [17], [20], [21], [25], local storage usage [1], [6], [25] and server storage overhead [6], [14], [15], [18]. Out of the many different proposed ORAM versions and improvements, PathORAM introduced by Stefanov et al. has a simple structure that offers an efficient trade-off between communication complexity and local storage [22], setting the foundation for further research on the application of ORAM in cloud storage services. Yuan et. al improved upon PathORAM by implementing data sharing scheme where additional users can read or write data if they have been granted permission by the data owner [27], providing the ORAM scheme the ability to support multiple clients which is a necessary property to implement high-demand cloud storage service functions such as file sharing. Wolfe et al. further improve upon PathORAM by providing additional properties useful for cloud storage services, namely support for resizing the server capacity and packing multiple files in single blocks which ensures that the cloud storage space is not wasted [26]. There are several design details of PathORAM that have not been specified in the original paper, such as the method of block encryption and the initialisation of empty blocks in the server storage. Gordon et al. compared different solutions to these unspecified design details in the context of cloud storage services and provided the most efficient solutions [8].

Besides regular cloud storage, the ORAM algorithms with efficient communication complexity and storage efficiency can also be applied to other practical fields. The most obvious of these is the field of cloud computing, where the cloud server storage contains an (encrypted) ORAM structure that can be loaded and executed by specialised processors on the server itself. Maas et al. introduce an improvement on the PathORAM algorithm together with a specialised secure processor named Phantom which can read and write to the ORAM structure [16]. The ORAM structure contains the sensitive data to be processed as well as the code necessary to process this data, which can be executed by the secure processor. After the result of the calculations have been stored in the ORAM structure, it can be accessed again by the client. Because most ORAM operations in cloud computing are performed on the server itself without communicating with the client, regular properties such as communication complexity can not be solely used to evaluate their effectiveness. Rather it is the overhead of the processor securely accessing the ORAM structure and executing instructions that better represents the efficiency of the ORAM schemes used in cloud computing [24].

Cloud computing is an important part in the internet-of-things(IoT) field, allowing devices to obtain results of complex calculations or results calculated with the data collected by multiple devices. Because of the design of IoT devices, they often posses low calculation abilities and small storage capacity. This results in requirements for memory usage and calculation complexity while communicating with cloud computing services. For this purpose Huang et al. introduce the ThinORAM algorithm, which has very low calculation complexity for the client side, low communication complexity between client and server and has low demands on the client-side storage [10].

4. RESEARCH QUESTIONS AND METHODOLOGY

Before any measurements and tests can be done we must first set up the client-server infrastructure. The server is a simple structure supporting a specific maximum storage. During initialisation it creates one (empty) file for each block until the total size matches the maximum storage. The server also supports two separate operations, one is to read a specific block which simply returns the value from the related file, while the write operation writes the data sent and returns the previously stored file content. The client performs AES-128 CTR-mode encryption over any block it communicates with the server, so that the server is unable to decrypt any block that has been sent.

In order to answer the research question introduced in section 1.1, we need to analyse the answers of the 3 related research sub-questions that are introduced in the following subsections. Subsection 4.1 elaborates on the method used to convert the files into proper addresses/blocks that can be stored using ORAM, while subsection 4.2 elaborates on the ORAM implementations that are used to access these addresses/blocks on the server.

4.1. File conversion

Usually cloud storage services can store files (encrypted or plaintext) regardless of their sizes. However, in ORAM the size of the blocks that can be stored are always fixed to a specific size. Therefore in order to properly implement ORAM solutions in cloud storage services we must first find a method to map the variable sized files to blocks of a fixed size.

The details of this method can affect the efficiency of the final solution. For example, every file requires at least one block of storage space, if the blocksize is much larger then the file stored it will be inefficient. At the same time other file properties (such as file hash) can also be used in order to improve the method for file conversion.

Research sub-question 1: Which file conversion method is most suitable for a cloud storage service?

In order to find the best solution suitable for the problem described, we will need to try several different file conversion methods and measure their efficiency. In order to answer the research sub-question and select the most suitable file conversion method we will measure several quantifiable properties of each method, and use those properties to judge which method is the most suitable for cloud storage services. The details of this approach are further described in section 5.

4.2. ORAM implementation

In order to measure the efficiency of several different ORAM implementations we will introduce a specific situation for which they will be tested. The cloud storage service(server) only provides the most basic functionality of reading and writing blocks, while the client-side itself runs different ORAM implementations with various parameters to test the efficiency. The most important properties to measure the efficiency of the ORAM implementations are the network communication usage as well as the local storage usage, as these directly affect the practical use of cloud storage services.

Research sub-question 2: How efficient is the communication and local storage usage of the ORAM implementations?

These tests will be run using the file conversion options that have been found as an answer to research sub-question 1, and it will test three different ORAM implementations which are Square-root ORAM, Trivial bucket ORAM and PathORAM as described in section 2.2. In order to quantify the efficiency of network communication we measure the number of network requests between the client and the server, while the local storage usage is quantified by the sizes of the local cache as well as the index file. Because a larger local storage can reduce the communication complexity (and vice versa), we will need to measure both properties together to accurately asses the effectiveness of an ORAM implementation. The details of this measurement are further described in section 6.

Although the communication complexity and local storage usage are the most fundamental properties to consider for the usage of ORAM in cloud storage services, each ORAM implementation has its own properties that may affect the usage in this scenario. For example, PathORAM has a flexible shelter that can accommodate blocks which no longer fit in the server storage. However, because of these additional properties the communication efficiency and local storage capacity of these protocols may be affected. In order to be able to make a fair comparison between various implementations, we will keep their requirements simple and mention additional properties of the implementations separately as the third research sub-question.

The requirements of the ORAM implementation we test are the following:

1. Only a single client needs to use the cloud storage.
2. The server will not perform malicious inserts, deletes or updates. (therefore integrity checks are unnecessary)

Research sub-question 3: Which ORAM implementations have additional properties that can affect the implementation in cloud storage services?

In order to account for the influence of additional properties of the ORAM implementations on the cloud storage services we will analyse the detailed results of the ORAM measurements, and draw conclusions about how these results would affect cloud storage services. At the same time, we will also use the theoretical background knowledge to argue how this affects the usage of the implementations in cloud storage services.

5. FILE MEASUREMENT

In order to find an answer to research sub-question 1, as described in section 4.1, we will create a set-up which allows us to objectively measure the effectiveness of file conversion methods.

Research sub-question 1: Which file conversion method is most suitable for a cloud storage service?

In order to measure changes to the file system we will develop a python monitoring script that detects any additions, updates or deletions of files in a specific directory. This directory represents the file system that should be synchronised with the cloud storage service (server), and the contents should be converted to block addresses to be used by the ORAM protocol. This monitoring script will interact with a python file-conversion script and notify the file-conversion script of any changes to the directory. The file-conversion script will keep a local index file containing a record of the block addresses in which a file is stored, and update this record every time the monitoring script notifies it of any changes. The relation between the monitoring and file-conversion scripts are shown by figure 6.

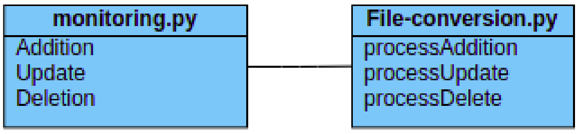


Figure 6: Class interface for file conversion

There will be only one file-conversion script, yet this script can accept different parameters which influence the way it works. By testing different options of the same script we can objectively quantify which options are best suited for cloud storage services. The following options are implemented in our code.

1. Block size, this value determines the minimum file size as well as the number of blocks that a (large) file needs to be split into.
2. File hash, if enabled the file hash is stored in the index file and can be used to detect whether a deleted file is moved to another location rather then deleted. If so, the content does not need to be re-uploaded to the server and we only need to adjust the name in the index file.
3. Block hash, if enabled the index file will record a hash of every block of the file. This is used to determine which blocks have been changed when a file is updated so that only changed blocks need to be uploaded to the server. Because of the implementation, this option can only be used together with the file hash option.
4. Extends, instead of enumerating every block that a file consists of in the index file we simply record the range, saving local storage space. E.g.: [1,2,3,6,7,8] will be recorded as [[1,3],[6,8]]

In order to properly quantify which options are best suited we need to use a generalised approach and measure the results. We describe the method we use to measure the file conversion method in more detail in subsection 5.1. The results of these measurements are described in subsection 5.2, and we compare these results in subsection 5.3 in order to determine which method is more suitable for cloud storage services, answering research sub-question 1.

5.1. Measurement/Experimental setup

In order to measure the effectiveness of the various options on the file conversion method we have created a script dedicated to measuring the effectiveness of file conversion, namely: measure-file-conversion.py. This script will be run with different options in order to judge their influence on the file conversion.

This script maintains an initially empty local directory, which is used by the script to add and change files according to the step that is being measured. Then the script will trigger the monitoring script that parses the changes in the local directory in order to update the index file. During this process, the measurement script will measure various properties in order to judge the effectiveness of the used options. The following properties are measured by the script during every step:

1. Time used by the monitoring script to find the changes to the file system and adjust the index file.
2. The size of the local index file. This property represents local storage usage.
3. The number of blocks that need to be updated during this step, which means that the blocks need to be uploaded to the server using ORAM. This property represents communication complexity.
4. Storage efficiency, which is calculated by dividing the total file size by the total size of all allocated blocks.

In order to ensure that the measurement accurately reflects the way in which cloud storage services can be used, the measurement script has 7 different steps to be measured. Each step performs a specific operation on the files in the directory so that the efficiency of these operations can be accurately measured. At the same time these steps also operate on files of various size, which reflect the fact that storage services often contain both small and large files. The following 7 steps are performed and measured by the script:

1. **Initialisation**, this step creates files of various sizes and then initialises the local index file. The following files will be created during this step:

    1000 small files, ranging from 1 byte to 1000 bytes.

    1000 medium files, ranging from 1KB to 1000KB

    19 large files, ranging from 1MB to 9MB, and from 10MB to 100MB in 10MB increments.

2. **Add**, this step creates files to be added to the directory. The following files will be created during this step:

    1000 small files ranging from 1 byte to 1000 bytes.

    1000 medium files ranging from 1KB to 1000KB

    6 large files of respectively 1,3,5,10,25,50 MB

3. **Move**, this step moves all the files created in step 2 and moves them to another directory.
4. **File reduction**, this step removes half the content of all files created during step 1 with an even file size.

   Because the larger files between 20MB and 100MB in size can all be considered as even, we make the distinction that only the following files are included in this step: 20MB, 40MB, 60MB, 80MB and 100MB.
5. **File increase**, this step doubles the content of all files created during step 1 with an odd file size.

   For the reasons provided in step 4, we include the following files in this step: 30MB, 50MB, 70MB, 90MB
6. **File change**, this step changes a single byte of each file moved during step 3.
7. **Deletion**, this step removes all files currently in the directory.

5.2. Results

The results of the measurement are included in appendix A, in this section we present a selection of these measurement results and draw conclusions about the influence of the four options on the measured properties.

The block size is the only option that influences the storage efficiency. Figure 7 illustrates the minimum and maximum storage efficiency measured in the different steps of the measurement, and compares it to the block size used as an option. The figure shows that the larger the block size becomes, the lower the storage efficiency will be.

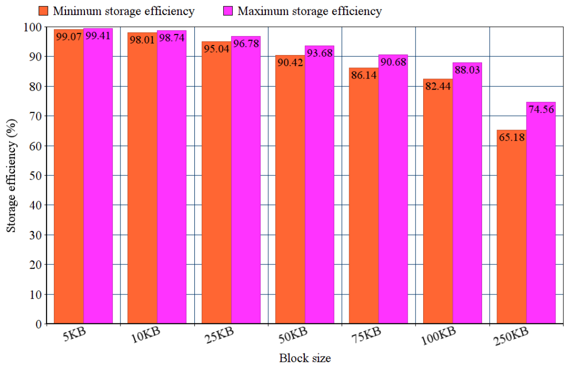


Figure 7: Measurement results comparing block size to storage efficiency. The orange bar represents the minimum storage efficiency, while the purple bar represents the maximum storage efficiency measured during any step of the measurement.

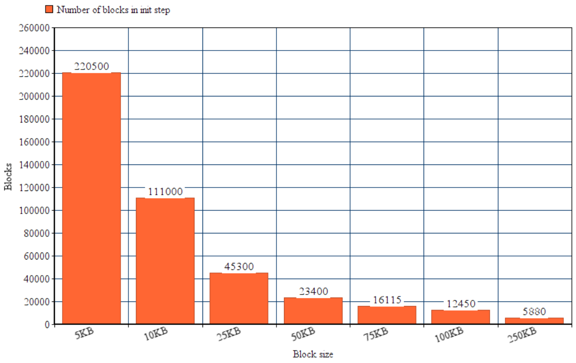


Figure 8: Measurement results comparing block size to the number of blocks measured during the *Init* step.

The block size influences the number of blocks that are used to store the files and their changes. In order to understand how this influence is expressed, we need to understand that this is also influenced by the options file hash and block hash in the steps *Move*, *File reduction*, *File increase* and *File change*. Therefore we display the correlation between block size and the number of blocks in figure 8 in the *init* step. This figure shows that the block size is inversely correlated to the number of blocks, as the increase in block size will reduce the number of blocks used. Because the measured results are dependant on the distribution of the file size of the files used to perform measurements, the measurement results may differ when files with significantly different sizes are used.

The index file enumerates every block ID that is in use when the extends-option is False, and when using the block hash option there will be a hash recorded for every block in the index file. Because the block size affects the number of blocks used, it will also affect the size of the index file, provided that the extends option is false or block hash is true. Figure 9 displays the relationship between the block size and the size of the index file during the *init* step, with all options as either false or true. From this we can see that a small block size will require a much larger index file then a larger block size, however the differences will become much smaller after the block size reaches a certain value (about 50KB). At the same time we see that the difference between all options as true instead of false increases the size of the index file with a factor between 8 and 4. The larger the block size becomes, the smaller the difference in index file size will be when comparing the measurements with either all options equal to true or false.

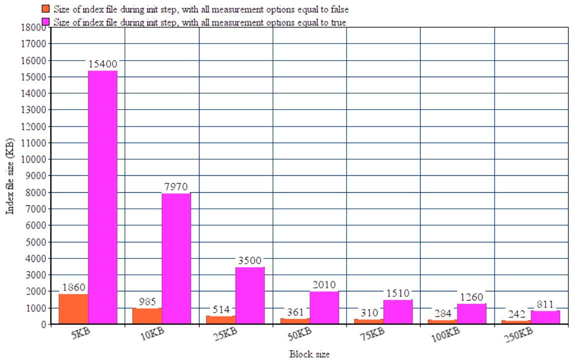


Figure 9: Measurement results comparing block size to index file size during the *Init* step. The orange bar represents results measured with all options equal to false, while the purple bar represents results measured with all options equal to true.

Block size not only affects the file size, but also the execution time measured during the steps in the same way. When extends is false or block hash is true, increasing the block size will reduce the execution time. Figure 10 displays the relationship between the block size and the execution time of the *move* step, with all measurement options equal to false. This figure shows that the increase of the block size will reduce the execution time, however after the block size reaches 50KB the differences will become much smaller.

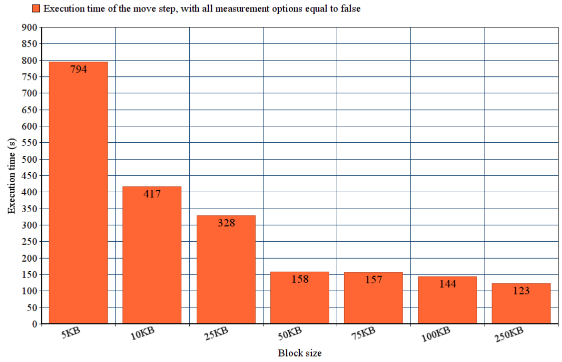


Figure 10: Measurement results comparing block size to execution time during the *Move* step, with all other measurement options equal to false.

The extends option reduces both the execution time and index file size of every step. The smaller the block size is, the larger the reduction effect of the extends option will be. After the block size exceeds 100KB the reduction effect of the index file size may be negligible, while the execution time will actually increase.

The file hash option increases both the execution time and index file size during every step. However, enabling this step also ensures that during the *move* step no blocks need to be transmitted to the server. This option is equal to expending additional execution time and local storage space in order to reduce communication complexity for a specific operation (moving files). The increase in both execution time and the index file are quite reasonable, roughly between 10% and 50% depending on the specific step and measurement.

The block hash option increases both the execution time and index file size significantly during each step. However, enabling this option also ensures that during steps that change file content the amount of blocks that need to be transmitted to the server will be significantly reduced. The increase in both time and index file size is highly dependant on the block size. If the block size is low the increase may reach up to 10 times the original, while with a high block size the increase may be about 50%-100%.

5.3. Analysis

In order to answer research sub-question 1 and find out which file conversion method is most suitable for cloud storage services, we must first consider the usage scenario. Considering that we use the cloud storage service specifically for storage purposes, we must ensure that the storage efficiency is not too low. We use a storage efficiency of 90% or higher as benchmark, because such a value ensures an acceptable storage overhead while most of our measurement results conform to it. In order to conform to our benchmark, it requires that the block size cannot be higher then 50KB.

At the same time, because of the expensive nature of ORAM communication we must reduce the communication complexity as much as is feasible. This means that we can also rule out block sizes which are too low, leading to a higher number of blocks and therefore communication complexity. Therefore it is most suitable to use a block size of exactly 50KB, as it has relatively good storage efficiency, low execution time and small index file. At the same time, we also enable the extends option as it only provides benefits and has no drawbacks, leading to a decreased execution time and index file size.

In order to further reduce the communication complexity, we will enable both file hash and block hash options. This ensures that during file operations only the blocks that have been changed will need to be uploaded again, saving lots of communication during regular cloud storage service operations such as editing files. This comes at the expense of significantly increasing both the execution time and the index file size. However, it is estimated that the increase in execution time is much less then transmitting these additional blocks using ORAM would take, as that is limited by the network communication speed. And the increase in the index file is estimated to be roughly equal to about 0.2% the file contents stored in the cloud storage service.

Our measurement results are all measured based on a setup where the files have a mixed composition of small, medium and large files. Therefore our analysis and solution are also best suited for cloud storage services where the files stored have a similar composition. If the file composition is significantly different we would need to re-run the tests with different files in order to obtain accurate measurement results, but based on the results and analysis of our current measurements we can still estimate how such compositions would change our analysis. If the files are all large in size, we could use much larger block sizes to store the data without sacrificing storage efficiency. This will result in a much smaller number of blocks, thus reducing both communication complexity as well as local storage usage. Because the files are all large the importance of block hash is even more significant, as it ensures that when a file is changed not all its content needs to be updated. If the files are all very small in size, the current block size must be reduced significantly in order to ensure that the storage efficiency is not too low. With files of a small size there will be few blocks per file, reducing the need for block hashes as there will be little overhead when all blocks of a changed file needs to be updated. At the same time, the lack of block hashes can ensure that the increase in local storage due to the large number of blocks is limited.

6. ORAM MEASUREMENT

There are two research sub-questions related to ORAM implementations posed in section 4.2. In order to answer these we introduce an experimental setup in which we measure the efficiency of the three different ORAM implementations: Square-root ORAM, PathORAM and Trivial ORAM.

The experimental setup uses the file conversion parameters found as answer to RQ 1 while testing the different ORAM implementations. Every time the file-conversion script described in section 5 needs to upload a block to the server, it triggers the client to upload the block. The client then invokes the specific ORAM script to performs various read and write operations to the server in accordance with the algorithm. In order to ensure that our measurements are not affected by network communication and potential delays, the communication between the client and server will happen through code rather than real network connections.

By comparing the measured results we can determine the ratio between communication complexity and local storage usage of the various ORAM implementations, answering RQ 2. In order to answer research sub-question 3 we do not need to perform additional measurements. By analysing the results of the measurement and the theoretical knowledge of the ORAM implementations, we can learn the influence of additional properties on cloud storage services.

Research sub-question 2: How efficient is the communication and local storage usage of the ORAM implementations?

Research sub-question 3: Which ORAM implementations have additional properties that can affect the implementation in cloud storage services?

Besides using the fixed file conversion parameters during the experiment, some ORAM implementations also require additional parameters. In order to measure their influence on the experiment, we will perform the measurements with various values. The following options are implemented in our code.

1. Blocks per bucket, this option is used in both Trivial ORAM and PathOram and is used to determine how many blocks are included in a single bucket.
2. Eviction rate, this option is only used by Trivial ORAM and is used to determine how many blocks should be evicted after every access operation.
3. Server storage capacity, this option is used by all ORAM implementations and refers to the maximum storage of the server. This property can be used to judge how the load of the database can influence its effectiveness and properties. At the same time, this parameter will also affect how often Square-root ORAM will perform end-of-epoch operations.

We describe the server storage capacity in terms of number of blocks rather then the usual data storage measurement units such as bytes, which is usually used to describe storage capacity. This is because some ORAM algorithms require a small number of bytes to be added to each block as a means of identification, and at the same time it also simplifies the description and analysis of our measurements. The specific values of the server storage capacity are dependant on the algorithm that is being tested, as both PathORAM and Trivial ORAM require the server storage to represent a binary tree containing buckets, and the Square-root ORAM requires the server storage to have a shelter of exactly the square-root of the server storage capacity. This means that the server storage must conform to the following requirements. For the binary tree algorithms it must conform to $(2^X - 1) * Z$ where Z equals the number of blocks per bucket and X can be any integer, and for Square-root ORAM it requires the server capacity to have an integer as square root. In order to make a fair comparison between measurements we require the server capacity to have similar values. For this reason the options block per bucket and server storage capacity we measure will be limited to values that result in similar server storage capacities. The specific values of the storage capacity as well as other parameters used in the measurements can be found in the table displaying measurement results in appendix B.

A more detailed description of the specific measurement method is described in subsection 6.1.

In the papers and background knowledge of the ORAM schemes, sometimes the details of specific situations are not addressed. In order to properly test these schemes we had to write code that implements the ORAM algorithms, this code should also account for these situations in a way that does not compromise the ORAM security definition.

During the eviction step of Trivial ORAM we read a block from a parent bucket and write it to the corresponding child bucket. However, if the child bucket is already full we cannot write the block to it. In this case we simply store the block and continue to finish the eviction operation. After the eviction operation has finished we simply perform a new access operation, writing the stored block to the root node according to the original algorithm. Only by doing this we can ensure that the network traffic does not deviate from the traffic during normal operation, so that third-party observers cannot learn that the child bucket is full.

During the access operation of the PathORAM we always read all buckets leading from the root node to the child leaf corresponding the position of the address written or read. However, the algorithm does not describe what to do when a new address is added and thus does not have a previous position to determine which path to read. In this case we simply assign a random value as temporary position in order to ensure that the network traffic follows the normal pattern, ensuring that third-party observers cannot learn that this address is newly added.

6.1. Measurement/Experimental setup

In order to measure the implementations of ORAM algorithms, we have created a script dedicated to measuring the ORAM implementation, namely: measure-oram.py. This script will be executed with different ORAM implementations and options to assess their effectiveness and other properties.

This script maintains a local directory, which is used as a local storage directory to be uploaded to the server using ORAM. The measurement script will add a number of files to this directory, and then triggers the monitoring script to parse the changes in the directory and invoke the ORAM script to write the affected blocks to the server. Because this experiment does not need to account for storage efficiency, the script will only write files with a lengths that are multiple times the block size. During this process the following properties will be measured in every step.

1. The number of network requests made by the ORAM scheme.
2. The size of the local index files, both for the file conversion and ORAM.
3. The maximum size of the local shelter during the execution of the ORAM algorithm.
4. The server load, which is the number of blocks uploaded divided by server capacity. This property may affect the efficiency and workings of the ORAM algorithms.
5. Execution time, separated in two parts. Namely, the execution time of the file conversion part, and the execution time of the ORAM algorithm and communication.
6. During the Square-root ORAM measurements we will also count how often the end-of-epoch has been triggered. At the same time we will also record how many network requests are performed during the end-of-epoch as well as its execution time.
7. During the Trivial ORAM measurements we will count how often eviction operations fail, and the evicted blocks have to be re-added to the root bucket.

Because the various use cases of file manipulation have already been covered by the file conversion measurement, this ORAM measurement only needs to cover the addition and reading of files. That means that this script only needs the following steps in order to cover all necessary measurements.

1. **Initialisation**, this step adds an amount of blocks to the local directory to ensure that the server load will read a certain standard. This is to ensure that the next steps will all be measured under the server load required.
2. **Add**, this step will add 50 files, each containing 10 blocks, to the local directory. In this step we can measure the properties while a server has a certain load.
3. **Increment**, this step will append a total of 500 blocks to randomly selected files that were added in step 2.
4. **Read**, this step will read all files previously added during step 2 from the server, and compare whether the results are still the same as those stored in the local directory.

During the measurements we use the initialisation step to partially load the server storage, so that subsequent steps measure their effectiveness under this load. Because we need to run a large number of tests for the different ORAM implementations, we first run all tests without adding any load at all, effectively omitting the initialisation step. After we have run all tests without any load, we will perform measurements with the initialisation steps of different loads only for a limited number of different options. These measurements will only be performed with the options that yield significant results for our research, so that we can determine the influence of the increased load on the measured properties. This approach ensures that we do not perform lengthy measurements that provide little to none additional information.

6.2. Results

The results of the previously described measurements are included in appendix B, in this section we present a selection of these measurement results and draw conclusions about the influence of the 3 options and the server load on the measured properties of the three different ORAM algorithms. We use the results displayed in the appendix to draw conclusions based on the influence of the options on the measured properties of each ORAM algorithm, while we use additional figures displaying representative values of each algorithm in order to compare the various algorithms.

The first conclusion we can draw from the analysis is about the differences between the three steps, *Add*, *Increment* and *Read*. The first two steps write 500 blocks to the server each, while the last step reads 1000 blocks from the server. From the results we can see that the number of network requests performed are directly related to the number of access operations requested. This means that the network requests measured during steps *Add* and *Increment* are the same for all algorithms, except for cases in which it is influenced by specific properties of this algorithm. The only influence on the number of network requests for the Square-root ORAM is how often the end-of-epoch is triggered, which may differ between the different steps. While the only influence for the Trivial ORAM algorithm is how often the eviction step has failed, as this is no different from performing an additional access operation. The measurements show that for all three algorithms the number of network requests will increase if either the server capacity, bucket size or the eviction rate increases. The only exception to this is when in Trivial ORAM the increase of these options results in fewer failed evictions, causing the total number of network requests to be reduced. At the same time we can also conclude that the number of network requests is not influenced by the server load for all algorithms except for Trivial ORAM, where the server load increases the number of failed evictions and thereby the number of network requests. Figure 11, displayed on the next page, compares the number of network requests between the three ORAM algorithms. It shows that while the number of requests of Square-root ORAM and PathORAM are quite similar, the number of requests made by Trivial ORAM is roughly 10 times larger.

The local storage usage of the ORAM implementation consists of three different components, namely the local file index, the local ORAM index, and the shelter itself. Out of these three components, the size of the local file index is the most straightforward, as it is independent of the ORAM algorithm and options used. The size of the local file index is linearly correlated to the amount of files and blocks that have been written to the server. For each measurement without any initial server load we see that the local index file size after the step *Add* equal 44KB, and for every measurement with an initial server load higher then 0% we see that the local index file size during the *Add* step is about 44KB then the file size during the *Init* step. At the same time the measurements of the *Init* step with various loads also show that the size of the local file index is linearly correlated with the load imposed during this step. The size of the local ORAM index differs per algorithm. For Square-root ORAM the local ORAM index is dependant only on the server capacity and will not change in size regardless of the server load. The local ORAM index file for Trivial ORAM and PathORAM records the same information and are of the same size, except for the negligible increase in file size when PathORAM needs to record information about the blocks stored in the shelter. Their ORAM index file size is not dependant on the value of the eviction rate, but it is related to the server capacity as well as the bucket size. An increase in bucket size will slightly reduce the size of the ORAM index file, while higher server capacity will result in an increase in file size. Similar to the local file index size, the size of the local ORAM index is also linearly correlated to the amount of blocks written to the server and therefore increases with the server load in PathORAM and Trivial ORAM. Figure 12, which is displayed on the next page, compares the local ORAM index file size between the three algorithms. As the results of the measurements of PathORAM are the same as those of Trivial ORAM, the Trivial ORAM results have been omitted from the figure. This figure indicates that the local ORAM index of Square-root ORAM is larger then the index of PathORAM and Trivial ORAM, especially when measuring results without any server load. The last component which represents the local storage usage is the shelter itself, and its value represent the number of blocks that are stored on the client each 50KB in size. Square-root ORAM has a fixed shelter size, equal to the square-root of the server capacity, and is not influenced by properties other then the capacity itself. Trivial ORAM has no concept of a shelter at all so the results always record the value 0 as basis for comparison, while the shelter size of PathORAM changes depending on the specific situation. In PathORAM a block will be added to the shelter if, during the access operation, it cannot be written to a bucket because it is full. This means that the shelter size will depend on how frequent a bucket is full, which is influenced by the various options. The results show that increased values of the bucket size lead to a significant decrease in shelter size, while an increased server capacity leads to a minor decrease in shelter size. At the same time an increase in server load will increase the shelter size, while also magnifying the influence of the other options on the shelter size.

The measurement results record the execution time during each step in two different parts, namely the execution time of the ORAM algorithm and execution time of the file conversion code. The results show that except for very few outliers, the execution time of the file conversion code during each step is less then 2% of the execution time of the ORAM algorithm itself. Because this value is negligible compared to the execution time of the ORAM scheme itself, we will only analyse the execution time of the ORAM algorithm. In all three algorithms the execution time will increase as the server capacity or bucket size does. At the same time an increase in eviction rate will increase the execution time in the Trivial ORAM

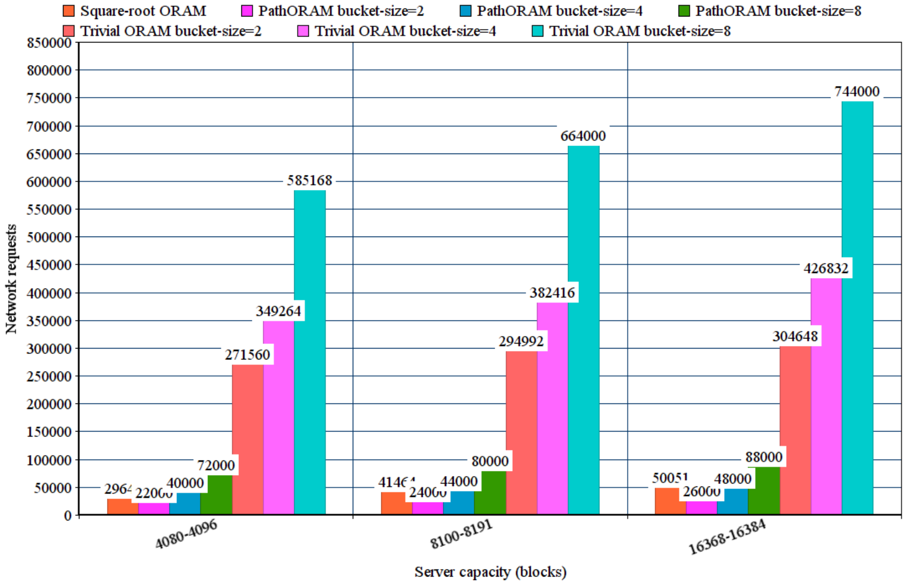


Figure 11: Measurement results comparing server capacity to network requests made during *Add* step, measurement results without any server load. Because the server capacity used during measurements of different algorithms and block sizes differ slightly, we display the results measured with similar server capacities in the same range. Note that all entries of Trivial ORAM came from measurements with eviction rate = 3.

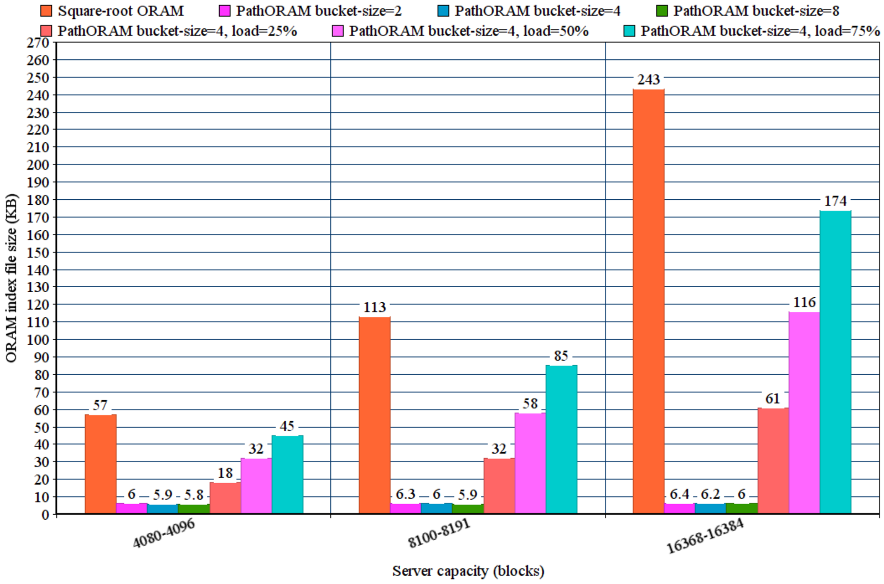


Figure 12: Measurement results comparing server capacity to local ORAM index file size, made during *Add* step. Note that the values of Trivial ORAM are the same as those of PathORAM, which is included in the figure.

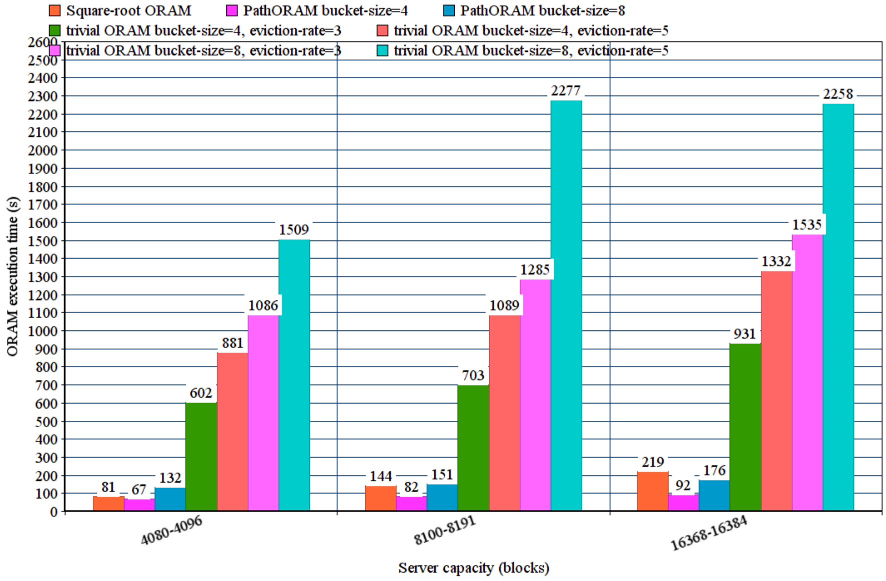


Figure 13: Measurement results comparing server capacity to ORAM execution time of the *Add* step, measurement results without any server load.

measurements, unless this increase causes a significant reduction in the number of failed evictions. The execution time of both Square-root ORAM and PathORAM are unaffected by the server load, while an increased load in Trivial ORAM will result in more failed evictions thus increasing the execution time. Figure 13 compares the execution time between the three algorithms. The figure shows that Square-root ORAM and PathORAM have similar execution times, while Trivial ORAM has a much longer execution time, up to 10 times that of the other two algorithms even without the influence of server load.

During the measurements of the Square-root ORAM we measure the additional properties related to the end-of-epoch, namely the number of times the end-of-epoch is invoked, the number of network requests made during this time as well as the execution time of the end-of-epoch invocations. From these results we can observe the following information, namely that during each step there is only one network request per access except for the additional network requests made during the end-of-epoch. At the same time, the amount of network requests made during a single end-of-epoch invocation is close to the server capacity plus the shelter size. The duration of the invocations of end-of-epoch during each step takes between 50% and 90% of the total ORAM execution time. The specific percentage is unaffected by server load, but a higher server capacity will lead to a higher percentage.

The measurements of the Trivial ORAM scheme also record another property, namely the amount of times that the eviction process failed. This means that the failed block needs to be added to the ORAM structure again, resulting in another access operation that requires a large number of network requests and execution time. The results of the measurements show that increasing the bucket size or the eviction rate will significantly decrease the number of failed evictions, while the server capacity has no influence on the number of failed evictions. However, regardless of the other options an measurement with an eviction rate of 1 will have a very high number of failed measurements compared to other eviction rate values.

Once the server load of Trivial ORAM exceeds 50% there will be a large chance that all leaf buckets of the data structure are full. The likelihood of this occurring will increase when the eviction rate is higher, as well as with the increase in server load. When all leaf buckets of the data structure are full, the evicted data will fail and thereby trigger the failed eviction process in which the data will be added to the root bucket again before triggering a new eviction process. This process can cause a large number of nested failed evictions which significantly slows down the access of blocks. Due to this sometimes access to a block may take over half an hour of execution time, which is too long for any practical use. Once this situation occurs we abort the measurement, and thus we only write the results of the steps that have been completed in the appendix. For the same reason we did not perform any measurements on Trivial ORAM with a load of 75%.

6.3. Analysis

In order to find the answer to research sub-quest 2, which is mentioned in section 4.2, we need to analyse the efficiency of the network communication and local storage usage of the three different ORAM algorithms. The results summarised in figure 11 show that the number of network requests made by Square-root ORAM and PathORAM are very similar, at least for results where the bucket size is 4. However, the number of network requests made by Trivial ORAM is almost 10 times as much as those made by the other two algorithms. Even if the bucket size increases, the difference in network requests between Square-root ORAM and PathORAM will be much smaller then the difference between PathORAM and Trivial ORAM. The results also show that neither Square-root ORAM nor PathORAM are influenced by the server load, but that a higher server load in Trivial ORAM will cause a significant increase in the number of network requests made.

As described in the previous section, we can compare the usage of local storage between the three algorithms by looking at the size of the local ORAM index file as well as the shelter. The results show that the size of the local ORAM index file is the same for the PathORAM and Trivial ORAM algorithms, except that the file of PathORAM can be slightly larger if there are additional blocks stored in the shelter. Comparing these two algorithms to Square-root ORAM, we see that the ORAM index file of Square-root ORAM is always larger regardless of the server load. Besides the ORAM index file, the other component necessary to judge the local storage usage of the algorithms is the shelter size. Square-root ORAM has a fixed shelter size which is equal to the square-root of the server capacity, while Trivial ORAM has no shelter at all. PathORAM has instead a flexible shelter size, which is generally much smaller then the shelter size of Square-root ORAM. However, if the bucket size is too low or the server load is too high, then the shelter size may become several times larger then that of Square-root ORAM. A shelter containing 90 blocks of 50KB each is already 4.5MB large, which is a size that occurs in many measurements of both Square-root ORAM and PathORAM. Because this is significantly larger than the size of the index files, we can claim that the efficiency of local storage usage is mainly defined by the size of the shelter. This leads to the fact that Trivial ORAM has the most efficient use of local storage, while the difference between PathORAM and Square-root ORAM depends on the shelter size of PathORAM.

In order to answer research sub-question 3 we need to find additional properties of the ORAM algorithms that have an influence on their usage in cloud storage services. The following paragraphs will describe the properties we found for each algorithm.

Square-root ORAM is an algorithm in which the majority of network requests and execution time are spend during the end-of-epoch. In fact, if you perform the access operation and do not trigger the end-of-epoch, then each access operation only costs one network request. Because of this Square-root ORAM is very efficient when handling files much smaller then the square-root of the server capacity, however it has a very high worst-case cost. This means that there may be significant delays when accessing a file, if the client encounters the end-of-epoch. The end-of-epoch is triggered precisely every time a fixed number (square-root of the server capacity) of blocks have been accessed. This means that Square-root ORAM is not very suited for active cloud storage services, where due to the clients actions access operations need immediate responses. However, this is no problem for cloud storage services such as Dropbox where a local directory is automatically synchronised to the server by a process running in the background. Due to the predictability of when end-of-epoch occurs, it is possible to trigger it on purpose at a desirable moment by randomly accessing sufficient blocks. In doing so it is possible to schedule an end-of-epoch at a moment when the device is not in use. This could be useful in situations where the amount of files accessed during the day is insufficient to trigger an end-of-epoch, so that the end-of-epoch itself can be triggered at a desirable moment such as at night. However there are few cases for cloud storage services where the number and size of the files that are accessed are few enough that end-of-epochs are never triggered during the day.

PathORAM is an algorithm which is not influenced by the server load in either the number of network requests nor in the execution time. At the same time, because of the nature of the algorithm, it has no large worst-case cost. Because each read or write operation requires the same amount of network requests and execution time, PathORAM is an algorithm with stable performance which is very applicable to cloud storage services where the client actively writes or reads files. At the same time, the maximum shelter capacity of PathORAM is not limited by the algorithm itself but only by the local storage capacity of the client. Because of this property, PathORAM can store additional blocks in the local shelter when the server load is full ensuring that blocks are not lost. This situation is very applicable to cloud storage services where usually a client pays for a limited amount of storage capacity, and only upgrades after it is completely full. In this case, the client can temporarily maintain the additional data in the PathORAM shelter until the server capacity has been upgraded.

Trivial ORAM is an algorithm which is negatively influenced by the server load. When the load of the server increases, so will the number of network requests as well as the execution time. This means that when the cloud storage service is in use, the more data is stored the less efficient Trivial ORAM will be. Although this can be partially mitigated by increasing the server storage capacity such that the server load is relatively less, this will ultimately lead to a large server-side storage overhead. Because cloud storage services often require payment or investment of appropriate hardware, a large server-side storage overhead is akin to having paid for storage capacity that can not be used. At the same time, increasing the server storage capacity will increase the number of network requests as well as the execution time of the Trivial ORAM algorithm required for each access operation or failed eviction. This means that only when the load is sufficiently large that the reduction of failed evictions due to the increase in server capacity offsets the increase in expenses for the access operation.

7. CONCLUSION

In order to answer the research question and judge what ORAM solutions are best suited for cloud storage services, we combine the answers of the sub-research questions described in sections 5.3 and 6.3.

The analysis of the file measurement section shows that it is most suitable for cloud storage services to reduce the amount of communication complexity at the expense of increasing the local storage usage. For this reason we use both file hashes and block hashes during the file conversion. At the same time we also enable the extends option, as this has no downsides and simple reduces the local storage usage. Based on the example directory used in testing the file conversion method we decided to use a block size of 50KB, which is a block size that ensures good storage efficiency and communication complexity. However, the block size chosen is dependant on the files used to perform the measurements. If the composition of the files used in cloud storage services significantly differs from the files used during the measurements, we suggest to adapt the block size to a value more suitable for the file composition.

The analysis of the ORAM measurement section shows the efficiency of the three algorithms in their local storage usage and communication complexity, as well as how their properties affect their usage in cloud storage services. Trivial ORAM has a very low local storage usage at the expense of requiring high communication complexity. While both Square-root ORAM and PathORAM posses a much more efficient communication complexity at the expense of increased local storage usage due to the size of their shelter. Cloud storage services are usually accessed by devices that posses reasonable amounts of local storage, and in situations where the client wants to access the files with as little delay as possible. Because of this, we can conclude that in cloud storage services it is better to reduce communication complexity at the expense of (linearly) increased local storage usage. This makes the Square-root ORAM and PathORAM best suited for cloud storage services, especially with regards to their usage of communication complexity and local storage.

In addition, because cloud storage services are used to store files it is common for them to have a significant server load. This means that Trivial ORAM, which is negatively influenced by server load, is not suited for usage in cloud storage services. Square-root ORAM is an algorithm with a very low best-case cost and very high worst-case cost, while PathORAM is a very stable algorithm in which each access has the same execution costs. This difference reflects the best use case scenario for each algorithm. Namely, PathORAM is best used in active cloud storage services in which the client requires stable and fast response to their access operations. This situation mainly occurs when the cloud storage services are used as external storage without possessing a local copy of the content, such as mobile phone usage where the local storage capacity is insufficient to store large amounts of files. Square-root ORAM, on the other hand, has variable response times depending on whether the end-of-epoch is triggered. This algorithm is best used in background processes where cloud storage services automatically synchronise local directories to the server, which is often used as an automated back-up.

7.1. Limitations and future work

Our ORAM measurements are performed with a server capacity ranging from 4080 blocks to 16384 blocks, this means that the maximum server storage that was used in our measurements is only 820MB. This ensures that our measurements can be performed in a reasonable amount of time, allowing us to perform measurements with many different options. The downside of this approach is that our result is not truly representative of cloud storage services with a much larger server capacity. Because of this we can only draw conclusions for such services based by extrapolating on the results we found and the influence of options on these results. Because of the large number of options in PathORAM and Trivial ORAM as well as the high execution time for performing load measurements, we limited the amount of measurements with initialised server load for these two algorithms. Although we do not have measurement results for all options, we performed the load measurements for some representative values so that we can draw conclusions based on inference.

Although our research is to find out which ORAM algorithm is best-suited for the usage in cloud storage services, the original versions of the ORAM algorithms we implemented lack many features that are required for providing functions that are often used in cloud storage services such as sharing files with third parties and integrity checks. Although there is plenty of research in adapting various ORAM algorithms to provide such functions, doing so would change the implementation of the algorithm. This means that the measurements we performed are no longer representative of the changed algorithm, and it requires additional research to verify their suitability for usage in cloud storage services.

Some aspects of the ORAM algorithms we used need to be carefully considered or adjusted in order to make it suitable for cloud storage services. For example, in Square-root ORAM the client-side shelter is used to store large amounts of changes before any updates are pushed to the server. Because of this, the algorithm is unsuitable for the usage of multiple clients unless the shelter will also be stored on the server. In Trivial ORAM when the server load exceeds 50% there will be a high chance that all leaf buckets are completely full. In this scenario it is guaranteed to trigger a failed eviction, which in turns causes a new access and eviction operation with the same high chance that all leaf buckets are full. Because of this reason, once the server loads exceeds 50% there will be a significant increase in nested failed evictions causing the execution of the algorithm to slow down significantly. If we wish to use Trivial ORAM in cloud storage services, we must first find another way to handle failed evictions without causing such nested loops. This may be solved by introducing a shelter that temporarily stores the failed evictions, and re-introduce the sheltered blocks into the binary tree in a way that does not compromise the ORAM security definition by showing deviations in the network traffic.

REFERENCES

[1] Dan Boneh, David Mazieres, and Raluca Popa. 2011. Remote Oblivious Storage: Making Oblivious RAM Practical. (03 2011).

[2] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (Nov. 1998), 965–981. DOI:`http://dx.doi.org/10.1145/293347.293350`

[3] Anders P. K. Dalskov and Claudio Orlandi. 2018. Can You Trust Your Encrypted Cloud? An Assessment of SpiderOakONE's Security. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 343–355. DOI: `http://dx.doi.org/10.1145/3196494.3196547`

[4] O. Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. Association for Computing Machinery, New York, NY, USA, 182–194. DOI: `http://dx.doi.org/10.1145/28395.28416`

[5] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473. DOI: `http://dx.doi.org/10.1145/233551.233553`

[6] Michael T. Goodrich and Michael Mitzenmacher. 2010. MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations. *CoRR* abs/1007.1259 (2010). `http://arxiv.org/abs/1007.1259`

[7] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. *Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation*. 157–167. DOI: `http://dx.doi.org/10.1137/1.9781611973099.14`

[8] Steven Gordon, Atsuko Miyaji, Chunhua Su, and Karin Sumongkayyothin. 2015. Analysis of Path ORAM toward Practical Utilization. In *2015 18th International Conference on Network-Based Information Systems*. 646–651. DOI: `http://dx.doi.org/10.1109/NBiS.2015.113`

[9] Thang Hoang, Ceyhun Ozkaptan, Gabriel Hackebeil, and Attila Yavuz. 2018. Efficient Oblivious Data Structures for Database Services on the Cloud. *IEEE Transactions on Cloud Computing* PP (11 2018), 1–1. DOI: `http://dx.doi.org/10.1109/TCC.2018.2879104`

[10] Yanyu Huang, Bo Li, Zheli Liu, Jin Li, Siu-Ming Yiu, Thar Baker, and Brij B. Gupta. 2020. ThinORAM: Towards Practical Oblivious Data Access in Fog Computing Environment. *IEEE Transactions on Services Computing* 13, 4 (2020), 602–612. DOI: `http://dx.doi.org/10.1109/TSC.2019.2962110`

[11] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*.

[12] Seny Kamara and Kristin Lauter. 2010. Cryptographic Cloud Storage. In *Financial Cryptography and Data Security*, Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, Josep M. Miret, Kazue Sako, and Francesc Sebé (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–149.

[13] Nasrin Khanezaei and Zurina Mohd Hanapi. 2014. A framework based on RSA and AES encryption algorithms for cloud computing services. In *2014 IEEE Conference on Systems, Process and Control (ICSPC 2014)*. 58–62. DOI: `http://dx.doi.org/10.1109/SPC.2014.7086230`

[14] Qiumao Ma, Jinsheng Zhang, Yang Peng, Wensheng Zhang, and Daji Qiao. 2016. SE-ORAM: A Storage-Efficient Oblivious RAM for Privacy-Preserving Access to Cloud Storage. In *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*. 20–25. DOI:`http://dx.doi.org/10.1109/CSCloud.2016.24`

[15] Qiumao Ma and Wensheng Zhang. 2019. Octopus ORAM: An Oblivious RAM with Communication and Server Storage Efficiency. *ICST Transactions on Security and Safety* 6 (04 2019), 162405. DOI: `http://dx.doi.org/10.4108/eai.29-4-2019.162405`

[16] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John

Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 311–324. DOI: `http://dx.doi.org/10.1145/2508859.2516692`

[17] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2014. Efficient Private File Retrieval by Combining ORAM and PIR. In *NDSS*.

[18] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM Revisited. Cryptology ePrint Archive, Report 2010/366. (2010). `https://ia.cr/2010/366`.

[19] Manish M. Potey, C.A. Dhote, and Deepak H. Sharma. 2016. Homomorphic Encryption for Security of Cloud Data. *Procedia Computer Science* 79 (2016), 175–181. DOI: `http://dx.doi.org/https://doi.org/10.1016/j.procs.2016.03.023` Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[20] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *Advances in Cryptology – ASIACRYPT 2011*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–214.

[21] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards Practical Oblivious RAM. *CoRR* abs/1106.3652 (2011). `http://arxiv.org/abs/1106.3652`

[22] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 299–310. DOI: `http://dx.doi.org/10.1145/2508859.2516660`

[23] Maha Tebaa and Said El Hajji. 2014. Secure Cloud Computing through Homomorphic Encryption. *CoRR* abs/1409.0829 (2014). `http://arxiv.org/abs/1409.0829`

[24] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: Oblivious RAM for Secure Computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 191–202. DOI: `http://dx.doi.org/10.1145/2660267.2660365`

[25] Peter Williams and Radu Sion. 2012. Single Round Access Privacy on Outsourced Storage. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 293–304. DOI: `http://dx.doi.org/10.1145/2382196.2382229`

[26] Nathan Wolfe, Ethan Zou, Ling Ren, and Xiangyao Yu. 2015. Optimizing Path ORAM for Cloud Storage Applications. (01 2015).

[27] Dandan Yuan, Xiangfu Song, Qiuliang Xu, Minghao Zhao, Xiaochao Wei, Hao Wang, and Han Jiang. 2018. An ORAM-based privacy preserving data sharing scheme for cloud storage. *Journal of Information Security and Applications* 39 (2018), 1–9. DOI: `http://dx.doi.org/https://doi.org/10.1016/j.jisa.2018.01.002`

[28] Samee Zahur, Xiao Wang, Mariana Raykova, Adria Gascon, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. 218–234. DOI: `http://dx.doi.org/10.1109/SP.2016.21`

[29] Wenying Zeng, Yuelong Zhao, Kairi Ou, and Wei Song. 2009. Research on Cloud Storage Architecture and Key Technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS '09)*. Association for Computing Machinery, New York, NY, USA, 1044–1048. DOI: `http://dx.doi.org/10.1145/1655925.1656114`

APPENDIX

A. FILE CONVERSION MEASUREMENT RESULTS

This section contains the results of the file conversion measurements, as described in section 5.1. Tables 3 to 9 contain the measurement results, each table represents a specific block size. Note that the raw measurement results from the code can be found in the git repository referred to in section 1.2.

5KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	1.2s	4.1s	7.7s	1.2s	4.2s	8.4s
	Blocks	220500	220500	220500	220500	220500	220500
	Index file	1.86M	2.01M	17.0M	232K	386K	15.4M
	Storage efficiency	99.41%	99.41%	99.41%	99.41%	99.41%	99.41%
Add	Time	332s	333s	763s	56s	65s	707s
	Blocks	120300	120300	120300	120300	120300	120300
	Index file	3.04M	3.35M	26.6M	477K	783K	24.0M
	Storage efficiency	99.24%	99.24%	99.24%	99.24%	99.24%	99.24%
Move	Time	794s	813s	1774s	207s	250s	1786s
	Blocks	120300	0	0	120300	0	0
	Index file	3.03M	3.34M	26.6M	464K	770K	24.0M
	Storage efficiency	99.24%	99.24%	99.24%	99.24%	99.24%	99.24%
File reduction	Time	203s	213s	471s	53s	64s	456s
	Blocks	58750	58750	900	58750	58750	900
	Index file	3.03M	3.33M	22.6M	900K	1.21M	20.5M
	Storage efficiency	99.07%	99.07%	99.07%	99.07%	99.07%	99.07%
File increase	Time	217s	228s	530s	47s	58s	527s
	Blocks	206700	206700	103900	206700	206700	103900
	Index file	3.39M	3.70M	30.0M	481K	787K	27.1M
	Storage efficiency	99.33%	99.33%	99.33%	99.33%	99.33%	99.33%
File change	Time	450s	467s	1119s	77s	95s	1167s
	Blocks	120300	120300	2006	120300	120300	2006
	Index file	3.39M	3.70M	30.0M	481K	787K	27.1M
	Storage efficiency	99.33%	99.33%	99.33%	99.33%	99.33%	99.33%
Deletion	Time	816s	827s	1329s	548s	563s	1358s
	Blocks	0	0	0	0	0	0
	Index file	2.97M	2.97M	2.97M	2.97M	2.97M	2.97M
	Storage efficiency	-	-	-	-	-	-

Table 3: File measurement results with a blocksize of 5KB

10KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	1.7s	7.4s	10.4s	1.8s	5.1s	7.8s
	Blocks	111000	111000	111000	111000	111000	111000
	Index file	985K	1.14M	8.72M	232K	385K	7.97M
	Storage efficiency	98.74%	98.74%	98.74%	98.74%	98.74%	98.74%
Add	Time	173s	201s	535s	50s	68s	289s
	Blocks	60900	60900	60900	60900	60900	60900
	Index file	1.69M	2.0M	13.8M	447K	782K	12.5M
	Storage efficiency	98.37%	98.37%	98.37%	98.37%	98.37%	98.37%
Move	Time	417s	499s	1108s	150s	200s	669s
	Blocks	60900	0	0	60900	0	0
	Index file	1.68M	1.98M	13.7M	464K	769K	12.5M
	Storage efficiency	98.37%	98.37%	98.37%	98.37%	98.37%	98.37%
File reduction	Time	114s	124s	305s	42s	62s	188s
	Blocks	29750	29750	950	29750	29750	950
	Index file	1.68M	1.98M	11.8M	666K	972K	10.8M
	Storage efficiency	98.01%	98.01%	98.01%	98.01%	98.01%	98.01%
File increase	Time	119s	183s	334s	40s	149s	210s
	Blocks	103700	103700	52450	103700	103700	52450
	Index file	1.86M	2.16M	15.4M	479K	785K	14.1M
	Storage efficiency	98.58%	98.58%	98.58%	98.58%	98.58%	98.58%
File change	Time	247s	291s	691s	72s	99s	423s
	Blocks	60900	60900	2006	60900	60900	2006
	Index file	1.86M	2.16M	15.4M	479K	785K	14.1M
	Storage efficiency	98.58%	98.58%	98.58%	98.58%	98.58%	98.58%
Deletion	Time	425s	463s	792s	300s	343s	556s
	Blocks	0	0	0	0	0	0
	Index file	1.44M	1.44M	1.44M	1.44M	1.44M	1.44M
	Storage efficiency	-	-	-	-	-	-

Table 4: File measurement results with a blocksize of 10KB

25KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	1.7s	4.6s	7.5s	2.1s	5.1s	7.6s
	Blocks	45300	45300	45300	45300	45300	45300
	Index file	514K	668K	3.78M	230K	384K	3.50M
	Storage efficiency	96.78%	96.78%	96.78%	96.78%	96.78%	96.78%
Add	Time	140s	116s	240s	82s	69s	175s
	Blocks	25260	25260	25260	25260	25260	25260
	Index file	910K	1.22M	6.08M	472K	778K	5.64M
	Storage efficiency	95.86%	95.86%	95.86%	95.86%	95.86%	95.86%
Move	Time	328s	270s	496s	187s	166s	384s
	Blocks	25260	0	0	25260	0	0
	Index file	897K	1.20M	6.06M	459K	765K	5.63M
	Storage efficiency	95.86%	95.86%	95.86%	95.86%	95.86%	95.86%
File reduction	Time	98s	87s	140s	60s	53s	113s
	Blocks	12350	12350	980	12350	12350	980
	Index file	897K	1.20M	5.28M	537K	843K	4.92M
	Storage efficiency	95.04%	95.04%	95.04%	95.04%	95.04%	95.04%
File increase	Time	97s	85s	178s	60s	52s	124s
	Blocks	41940	41940	21580	41940	41940	21580
	Index file	960K	1.27M	6.74M	473K	779K	6.25M
	Storage efficiency	96.34%	96.34%	96.34%	96.34%	96.34%	96.34%
File change	Time	204s	167s	314s	113s	99s	250s
	Blocks	25260	25260	2006	25260	25260	2006
	Index file	960K	1.27M	6.74M	474K	779K	6.25M
	Storage efficiency	96.34%	96.34%	96.34%	96.34%	96.34%	96.34%
Deletion	Time	322s	287s	350s	255s	192s	303s
	Blocks	0	0	0	0	0	0
	Index file	546K	546K	546K	546K	546K	546K
	Storage efficiency	-	-	-	-	-	-

Table 5: File measurement results with a blocksize of 25KB

50KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	2.8s	5.9s	8.2s	2.4s	5.5s	8.3s
	Blocks	23400	23400	23400	23400	23400	23400
	Index file	361K	514K	2.14M	229K	382K	2.01M
	Storage efficiency	93.68%	93.68%	93.68%	93.68%	93.68%	93.68%
Add	Time	73s	112s	197s	52s	83s	167s
	Blocks	13380	13380	13380	13380	13380	13380
	Index file	673K	979K	3.54M	470K	776K	3.34M
	Storage efficiency	91.95%	91.95%	91.95%	91.95%	91.95%	91.95%
Move	Time	158s	237s	413s	113s	177s	415s
	Blocks	13380	0	0	13380	0	0
	Index file	660K	966K	3.53M	457K	763K	3.33M
	Storage efficiency	91.95%	91.95%	91.95%	91.95%	91.95%	91.95%
File reduction	Time	49s	76s	127s	37s	58s	130s
	Blocks	6550	6550	990	6550	6550	990
	Index file	660K	966K	3.14M	495K	801K	2.97M
	Storage efficiency	90.42%	90.42%	90.42%	90.42%	90.42%	90.42%
File increase	Time	53s	79s	137s	39s	63s	147s
	Blocks	21340	21340	11290	21340	21340	11290
	Index file	692K	998K	3.87M	470K	775K	3.65M
	Storage efficiency	92.85%	92.85%	92.85%	92.85%	92.85%	92.85%
File change	Time	100s	153s	276s	73s	120s	216s
	Blocks	13380	13380	2006	13380	13380	2006
	Index file	691K	998K	3.87M	470K	775K	3.65M
	Storage efficiency	92.85%	92.85%	92.85%	92.85%	92.85%	92.85%
Deletion	Time	136s	198s	293s	115s	171s	217s
	Blocks	0	0	0	0	0	0
	Index file	278K	278K	278K	278K	278K	278K
	Storage efficiency	-	-	-	-	-	-

Table 6: File measurement results with a blocksize of 50KB

75KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	4.2s	7.0s	9.0s	3.5s	6.4s	9.0s
	Blocks	16115	16115	16115	16115	16115	16115
	Index file	310K	463K	1.59M	228K	382K	1.51M
	Storage efficiency	90.68%	90.68%	90.68%	90.68%	90.68%	90.68%
Add	Time	75s	91s	127s	57s	72s	115s
	Blocks	9431	9431	9431	9431	9431	9431
	Index file	594K	900K	2.70M	470K	775K	2.58M
	Storage efficiency	88.26%	88.26%	88.26%	88.26%	88.26%	88.26%
Move	Time	157s	179s	263s	118s	149s	234s
	Blocks	9431	0	0	9431	0	0
	Index file	582K	887K	2.69M	457K	762K	2.56M
	Storage efficiency	88.26%	88.26%	88.26%	88.26%	88.26%	88.26%
File reduction	Time	49s	55s	84s	38s	51s	79s
	Blocks	4629	4629	1002	4629	4629	1002
	Index file	581K	887K	2.43M	481K	786K	2.33M
	Storage efficiency	86.14%	86.14%	86.14%	86.14%	86.14%	86.14%
File increase	Time	49s	61s	90s	39s	53s	83s
	Blocks	14484	14484	7862	14484	14484	7862
	Index file	603K	909K	2.91M	468K	774K	2.78M
	Storage efficiency	89.49%	89.49%	89.49%	89.49%	89.49%	89.49%
File change	Time	101s	114s	177s	77s	113s	163s
	Blocks	9431	9431	2006	9431	9431	2006
	Index file	603K	909K	2.91M	468K	773K	2.78M
	Storage efficiency	89.49%	89.49%	89.49%	89.49%	89.49%	89.49%
Deletion	Time	129s	143s	187s	107s	150s	173s
	Blocks	0	0	0	0	0	0
	Index file	189K	189K	189K	189K	189K	189K
	Storage efficiency	-	-	-	-	-	-

Table 7: File measurement results with a blocksize of 75KB

100KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	4.9s	7.4s	10s	4.8s	7.4s	10s
	Blocks	12450	12450	12450	12450	12450	12450
	Index file	284K	438K	1.32M	228K	382K	1.26M
	Storage efficiency	88.03%	88.03%	88.03%	88.03%	88.03%	88.03%
Add	Time	67s	81s	119s	58s	69s	106s
	Blocks	7440	7440	7440	7440	7440	7440
	Index file	555K	861K	2.28M	469K	775K	2.19M
	Storage efficiency	85.02%	85.02%	85.02%	85.02%	85.02%	85.02%
Move	Time	144s	157s	245s	120s	134s	213s
	Blocks	7440	0	0	7440	0	0
	Index file	542K	848K	2.26M	456K	762K	2.18M
	Storage efficiency	85.02%	85.02%	85.02%	85.02%	85.02%	85.02%
File reduction	Time	43s	55s	80s	40s	48s	72s
	Blocks	3650	3650	995	3650	3650	995
	Index file	542K	848K	2.07M	473K	779K	2.00M
	Storage efficiency	82.44%	82.44%	82.44%	82.44%	82.44%	82.44%
File increase	Time	48s	56s	87s	43s	51s	77s
	Blocks	11050	11050	6150	11050	11050	6150
	Index file	558K	864K	2.43M	466K	772K	2.34M
	Storage efficiency	86.53%	86.53%	86.53%	86.53%	86.53%	86.53%
File change	Time	90s	110s	160s	79s	95s	149s
	Blocks	7440	7440	2006	7440	7440	2006
	Index file	559K	864K	2.43M	465K	771K	2.34M
	Storage efficiency	86.53%	86.53%	86.53%	86.53%	86.53%	86.53%
Deletion	Time	108s	128s	161s	97s	110s	154s
	Blocks	0	0	0	0	0	0
	Index file	144K	144K	144K	144K	144K	144K
	Storage efficiency	-	-	-	-	-	-

Table 8: File measurement results with a blocksize of 100KB

250KB block size	**Extends**	False			True		
	File Hash	False	True		False	True	
	Block Hash	False	False	True	False	False	True
Init	Time	12s	14s	16s	12s	13s	21s
	Blocks	5880	5880	5880	5880	5880	5880
	Index file	242K	396K	828K	226K	379K	811K
	Storage efficiency	74.56%	74.56%	74.56%	74.56%	74.56%	74.56%
Add	Time	64s	79s	106s	78s	78s	153s
	Blocks	3876	3876	3876	3876	3876	3876
	Index file	484K	790K	1.52M	462K	768K	1.50M
	Storage efficiency	69.33%	69.33%	69.33%	69.33%	69.33%	69.33%
Move	Time	123s	138s	195s	129s	138s	287s
	Blocks	3876	0	0	3876	0	0
	Index file	471K	777K	1.50M	449K	755K	1.48M
	Storage efficiency	69.33%	69.33%	69.33%	69.33%	69.33%	69.33%
File reduction	Time	42s	52s	72s	43s	55s	97s
	Blocks	1910	1910	998	1910	1910	998
	Index file	471K	777K	1.43M	455K	761K	1.41M
	Storage efficiency	65.18%	65.18%	65.18%	65.18%	65.18%	65.18%
File increase	Time	44s	53s	73s	46s	56s	107s
	Blocks	4868	4868	3058	4868	4868	3058
	Index file	478K	784K	1.57M	454K	760K	1.55M
	Storage efficiency	71.93%	71.93%	71.93%	71.93%	71.93%	71.93%
File change	Time	83s	102s	138s	84s	112s	205s
	Blocks	3876	3876	2006	3876	3876	2006
	Index file	478K	784K	1.57M	454K	760K	1.55M
	Storage efficiency	71.93%	71.93%	71.93%	71.93%	71.93%	71.93%
Deletion	Time	85s	99s	166s	85s	100s	182s
	Blocks	0	0	0	0	0	0
	Index file	64K	64K	64K	64K	64K	64K
	Storage efficiency	-	-	-	-	-	-

Table 9: File measurement results with a blocksize of 250KB

B. ORAM MEASUREMENT RESULTS

This section contains the results of the ORAM measurements, as described in section 6.1. Tables 10 and 11 contain the Square-root ORAM measurement results, tables 12 to 17 contain the PathORAM measurement results, and tables 18 to 31 contain the Trivial ORAM measurement results. Note that the raw measurement results from the code can be found in the git repository referred to in section 1.2.

Note that for load measurements of Trivial ORAM, we did not perform any measurements with a load of 75%. At the same time, due to bad performance of Trivial ORAM when the server load exceeds 50% some measurements were aborted before they were finished. In this case only the measurement results for steps that were completed are recorded in the results. More details about this can be found in sections 6.2 and 7.1.

SQRT - part 1	**Server capacity**	4096 blocks				8100 blocks	
	Server load	0%	25%	50%	75%	0%	25% %
Init	Network requests	0	67637	135277	202928	0	182271
	Local index file(oram)	57K	57K	57K	57K	113K	113K
	Local index file(file)	68B	91K	182K	273K	68B	180K
	Shelter size	64	64	64	64	90	90
	time(ORAM)	0s	156s	264s	447s	0s	710s
	time(File conversion)	0s	0.5s	1.3s	3.6s	0s	1.6s
	Server load	0%	25%	50%	75%	0%	25%
	end-of-epoch occurrences	0	16	32	48	0	22
	time(eoe)	0s	87s	124s	231s	0s	498s
	network requests(eoe)	0	66613	133229	199856	0	180246
Add	Network requests	29641	29640	29652	29643	41464	49658
	Local index file(oram)	57K	57K	57K	57K	113K	113K
	Local index file(file)	44K	135K	226K	317K	44K	224K
	Shelter size	64	64	64	64	90	90
	time(ORAM)	81s	73s	67s	73s	144s	195s
	time(File conversion)	0.2s	0.5s	1s	1s	0.7s	0.9s
	Server load	12.2%	37.2%	62.2%	87.2%	6.2%	31.2%
	end-of-epoch occurrences	7	7	7	7	5	6
	time(eoe)	51s	40s	29s	40s	90s	143s
	network requests(eoe)	29141	29140	29152	29143	40964	49158
Increment	Network requests	33806	33803	33797	33805	49666	41473
	Local index file(oram)	57K	57K	57K	57K	113K	113K
	Local index file(file)	79K	170K	261K	352K	79K	259K
	Shelter size	64	64	64	64	90	90
	time(ORAM)	95s	89s	72s	75s	187s	173s
	time(File conversion)	0.4s	0.7s	1.4s	1.2s	0.6s	0.9s
	Server load	24.4%	49.4%	74.4%	99.4%	12.4%	37.4%
	end-of-epoch occurrences	8	8	8	8	6	5
	time(eoe)	64s	51s	36s	41s	136s	120s
	network requests(eoe)	33306	33303	33297	33305	49166	40973
Read	Network requests	67630	67611	67632	67619	91122	91132
	Local index file(oram)	56K	57K	57K	57K	113K	113K
	Local index file(file)	79K	170K	261K	352K	79K	259K
	Shelter size	64	64	64	64	90	90
	time(ORAM)	202s	150s	130s	150s	372s	369s
	time(File conversion)	0.1s	0.2s	0.1s	0.2s	0.2s	0.1s
	Server load	24.4%	49.4%	74.4%	99.4%	12.4%	37.4%
	end-of-epoch occurrences	16	16	16	16	11	11
	time(eoe)	142s	80s	62s	82s	269s	265s
	network requests(eoe)	66630	66611	66632	66619	90122	90132

Table 10: ORAM measurement results of Square-root - part 1

SQRT - part 2	**Server capacity**	8100 blocks		16384 blocks			
	Server load	50%	75%	0%	25%	50%	75% %
Init	Network requests	372777	555066	0	523617	1065214	1597840
	Local index file(oram)	113K	113K	243K	243K	243K	243K
	Local index file(file)	359K	539K	68B	363K	727K	1.09M
	Shelter size	90	90	128	128	128	128
	time(ORAM)	1286s	1186s	0s	3343s	10098s	15371s
	time(File conversion)	3.4s	6.4s	0s	36s	94s	144s
	Server load	50%	75%	0%	25%	50%	75%
	# end-of-epochs	45	67	0	32	64	96
	time(eoe)	922s	1326s	0s	2772s	9017s	13765s
	network requests(eoe)	368727	548991	0	528521	1057022	1585552
Add	Network requests	41479	49663	50045	50051	50043	50049
	Local index file(oram)	113K	113K	243K	243K	243K	243K
	Local index file(file)	404K	583K	44K	408K	771K	1.14M
	Shelter size	90	90	128	128	128	128
	time(ORAM)	142s	165s	219s	577s	531s	585s
	time(File conversion)	1.2s	1.6s	0.3s	3.9s	49s	45s
	Server load	56.2%	81.2%	3.1%	28.1%	53.1%	78.1%
	# end-of-epochs	5	6	3	3	3	3
	time(eoe)	96s	122s	150s	511s	463s	517s
	network requests(eoe)	40979	49163	49545	49551	49543	49549
Increment	Network requests	49670	41462	66567	66565	66562	66565
	Local index file(oram)	113K	113K	243K	243K	243K	243K
	Local index file(file)	438K	618K	79K	442K	806K	1.17M
	Shelter size	90	90	128	128	128	128
	time(ORAM)	165s	149s	479s	1093s	586s	651s
	time(File conversion)	1.3s	1.9s	0.4s	5.9s	23s	33s
	Server load	62.4%	87.4%	6.1%	31.1%	56.1%	81.1%
	# end-of-epochs	6	5	4	4	4	4
	time(eoe)	120s	102s	408s	1024s	516s	581s
	network requests(eoe)	49170	40962	66067	66065	66062	66065
Read	Network requests	91132	91140	133140	133127	133129	133128
	Local index file(oram)	113K	113K	243K	243K	243K	243K
	Local index file(file)	438K	618K	79K	442K	806K	1.17M
	Shelter size	90	90	128	128	128	128
	time(ORAM)	320s	314s	682s	1332s	617s	630s
	time(File conversion)	0.1s	0.2s	0.1s	0.2s	0.1s	0.1s
	Server load	62.4%	87.4%	6.1%	31.3%	56.1%	81.1%
	# end-of-epochs	11	11	8	8	8	8
	time(eoe)	228s	221s	539s	1193s	472s	486s
	network requests(eoe)	90132	90140	132140	132127	132129	132128

Table 11: ORAM measurement results of Square-root - part 2

Pathoram - 1	**Bucket size (Z)**	1			2		
	Server capacity	4095	8191	16383	4094	8190	16382
Add	Network requests	12000	13000	14000	22000	24000	26000
	Local index file(oram)	6.7K	6.8K	6.8K	6K	6.3K	6.4K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	92	90	80	20	25	9
	time(ORAM)	21s	24s	27s	33s	40s	49s
	time(File conversion)	0.2s	0.2s	0.2s	0.2s	0.2s	0.2s
	Server load	12.2%	6.1%	3.1%	12.2%	6.1%	3.1%
Increment	Network requests	12000	13000	14000	22000	24000	26000
	Local index file(oram)	13K	13K	14K	12K	13K	13K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	155	151	150	23	25	18
	time(ORAM)	27s	32s	39s	32s	36s	50s
	time(File conversion)	0.5s	0.4s	0.4s	0.4s	0.4s	0.4s
	Server load	24.4%	12.2%	6.1%	24.4%	12.2%	6.1%
Read	Network requests	24000	26000	28000	44000	48000	52000
	Local index file(oram)	13K	14K	14K	12K	12K	13K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	156	162	154	25	35	30
	time(ORAM)	55s	63s	82s	65s	76s	98s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	24.4%	12.2%	6.1%	24.4%	12.2%	6.1%

Table 12: ORAM measurement result of Pathoram with bucket sizes 1 and 2, the init step is not displayed because all measurements are initiated with 0% basic load.

Pathoram - 2	**Bucket size (Z)**	4			8		
	Server capacity	4092	8188	16380	4088	8184	16376
Add	Network requests	40000	44000	48000	72000	80000	88000
	Local index file(oram)	5.9K	6.0K	6.2K	5.8K	5.9K	6.0K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	3	2	2	0	0	0
	time(ORAM)	67s	82s	92s	132s	151s	176s
	time(File conversion)	0.2s	0.2s	0.6s	0.2s	0.2s	0.2s
	Server load	12.2%	6.1%	3.1%	12.2%	6.1%	3.1%
Increment	Network requests	40000	44000	48000	72000	80000	88000
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	3	3	3	0	0	0
	time(ORAM)	68s	78s	87s	123s	154s	164s
	time(File conversion)	0.4s	0.4s	0.4s	0.4s	0.5s	0.8s
	Server load	24.4%	12.2%	6.1%	24.5%	12.2%	6.1%
Read	Network requests	80000	88000	96000	144000	160000	176000
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	4	3	3	0	0	0
	time(ORAM)	144s	157s	178s	244s	332s	395s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	24.4%	12.2%	6.1%	24.5%	12.2%	6.1%

Table 13: ORAM measurement result of Pathoram with bucket sizes 4 and 8, the init step is not displayed because all measurements are initiated with 0% basic load.

Pathoram - 3	**Bucket size (Z)**	16		
	Server capacity	4080	8176	16369
Add	Network requests	128000	144000	160000
	Local index file(oram)	5.6K	5.8K	5.9K
	Local index file(file)	44K	44K	44K
	Shelter size	0	0	0
	time(ORAM)	224s	258s	340s
	time(File conversion)	0.2s	0.2s	0.4s
	Server load	12.3%	6.1%	3.1%
Increment	Network requests	128000	144000	160000
	Local index file(oram)	11K	12K	12K
	Local index file(file)	79K	79K	79K
	Shelter size	0	0	0
	time(ORAM)	232s	261s	344s
	time(File conversion)	0.4s	0.6s	0.4s
	Server load	24.5%	12.2%	6.1%
Read	Network requests	256000	288000	320000
	Local index file(oram)	11K	12K	12K
	Local index file(file)	79K	79K	79K
	Shelter size	0	0	0
	time(ORAM)	452s	571s	657s
	time(File conversion)	0.1s	0.1s	0.1s
	Server load	24.5%	12.2%	6.1%

Table 14: ORAM measurement result of Pathoram with bucket size 16, the init step is not displayed because all measurements are initiated with 0% basic load.

Pathoram - 25% load	**Bucket size (Z)**	2	4			8
	Server capacity	8190	4092	8188	16380	8184
Init	Network requests	98256	81840	180136	393120	327260
	Local index file(oram)	27K	12K	25K	54K	25K
	Local index file(file)	182K	91K	182K	363K	181K
	Shelter size	37	5	3	3	0
	time(ORAM)	162s	151s	302s	959s	583s
	time(File conversion)	1.2s	0.6s	1.1s	8.4s	1.4s
	Server load	25%	25%	25%	25%	25%
Add	Network requests	24000	40000	44000	48000	80000
	Local index file(oram)	33K	18K	32K	61K	32K
	Local index file(file)	226K	135K	226K	408K	226K
	Shelter size	37	5	3	3	0
	time(ORAM)	40s	74s	74s	115s	142s
	time(File conversion)	0.7s	0.5s	0.6s	1.3s	0.8s
	Server load	31.1%	37.2%	31.1%	28.1%	31.1%
Increment	Network requests	24000	40000	44000	48000	80000
	Local index file(oram)	40K	25K	38K	68K	38K
	Local index file(file)	261K	170K	261K	442K	260K
	Shelter size	52	5	4	11	0
	time(ORAM)	41s	71s	74s	122s	144s
	time(File conversion)	0.8s	0.9s	0.8s	1.9s	0.9s
	Server load	37.2%	49.4%	37.2%	31.1%	37.2%
Read	Network requests	48000	80000	88000	96000	160000
	Local index file(oram)	40K	25K	38K	68K	38K
	Local index file(file)	261K	170K	261K	442K	260K
	Shelter size	53	5	4	12	0
	time(ORAM)	84s	152s	151s	264s	286s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	37.2%	49.4%	37.2%	31.1%	37.2%

Table 15: ORAM measurement result of Pathoram with initialisation load of 25%

Pathoram - 50% load	**Bucket size (Z)**	2	4			8
	Server capacity	8190	4092	8188	16380	8184
Init	Network requests	196560	163680	360272	786240	654720
	Local index file(oram)	54K	25K	52K	109K	51K
	Local index file(file)	363K	181K	363K	727K	363K
	Shelter size	78	4	5	5	0
	time(ORAM)	331s	298s	601s	2044s	1153s
	time(File conversion)	3.7s	1.4s	3.6s	42s	3.8s
	Server load	50%	50%	50%	50%	50%
Add	Network requests	24000	40000	44000	48000	80000
	Local index file(oram)	62K	32K	58K	116K	58K
	Local index file(file)	408K	226K	408K	771K	407K
	Shelter size	130	4	5	5	0
	time(ORAM)	45s	71s	73s	114s	138s
	time(File conversion)	1.2s	0.8s	1.5s	3.2s	1.2s
	Server load	56.1%	62.2%	56.1%	53.1%	56.1%
Increment	Network requests	24000	40000	44000	48000	80000
	Local index file(oram)	68K	38K	65K	123K	64K
	Local index file(file)	442K	260K	442K	806K	442K
	Shelter size	130	4	5	5	0
	time(ORAM)	50s	77s	72s	115s	136s
	time(File conversion)	1.4s	1.0s	1.3s	3.2s	1.4s
	Server load	62.2%	74.4%	62.2%	56.1%	62.2%
Read	Network requests	48000	80000	88000	96000	160000
	Local index file(oram)	68K	38K	65K	123K	64K
	Local index file(file)	442K	260K	442K	806K	442K
	Shelter size	132	9	5	5	0
	time(ORAM)	96s	131s	146s	226s	273s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	62.2%	74.4%	62.2%	56.1%	62.2%

Table 16: ORAM measurement result of Pathoram with initialisation load of 50%

Pathoram - 75% load	**Bucket size (Z)**	2	4			8
	Server capacity	8190	4092	8188	16380	8184
Init	Network requests	294816	245520	540408	1179360	982080
	Local index file(oram)	83K	38K	78K	167K	78K
	Local index file(file)	545K	272K	545K	1.09M	545K
	Shelter size	240	5	4	7	0
	time(ORAM)	571s	438s	897s	2584s	1698s
	time(File conversion)	8.3s	2.1s	6.1s	66s	7.5s
	Server load	75%	75%	75%	75%	75%
Add	Network requests	24000	40000	44000	48000	80000
	Local index file(oram)	91K	45K	85K	174K	84K
	Local index file(file)	589K	317K	589K	1.14M	589K
	Shelter size	377	38	20	12	0
	time(ORAM)	78s	62s	68s	104s	132s
	time(File conversion)	2.0s	1.1s	1.9s	14s	1.7s
	Server load	81.1%	87.2%	81.1%	78.1%	81.1%
Increment	Network requests	24000	40000	44000	48000	80000
	Local index file(oram)	98K	52K	92K	181K	90K
	Local index file(file)	624K	351K	624K	1.17M	624K
	Shelter size	563	229	64	15	0
	time(ORAM)	85s	65s	65s	100s	125s
	time(File conversion)	2.3s	1.2s	1.7s	3.2s	2.3s
	Server load	87.2%	99.4%	87.2%	81.1%	87.2%
Read	Network requests	48000	80000	88000	96000	160000
	Local index file(oram)	98K	52K	92K	181K	90K
	Local index file(file)	624K	351K	624K	1.17M	624K
	Shelter size	567	235	74	17	0
	time(ORAM)	208s	183s	129s	203s	246s
	time(File conversion)	0.2s	0.2s	0.2s	0.1s	0.1s
	Server load	87.2%	99.4%	87.2%	81.1%	87.2%

Table 17: ORAM measurement result of Pathoram with initialisation load of 75%

Trivial - 1	**Bucket size**	1					
	Server capacity	4095			8191		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	303232	253586	280896	351400	278658	326740
	Local index file(oram)	6.2K	6.2K	6.2K	6.4K	6.4K	6.4K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	545s	469s	523s	705s	566s	673s
	time(File conversion)	0.2s	0.2s	0.2s	0.2s	0.4s	0.2s
	Server load	12.2%	12.2%	12.2%	6.1%	6.1%	6.1%
	Failed evictions	2796	731	412	3014	733	461
Increment	Network requests	443072	361530	395164	406200	362278	379440
	Local index file(oram)	12K	12K	12K	13K	13K	13K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	836s	679s	743s	834s	757s	797s
	time(File conversion)	0.5s	0.4s	0.4s	0.4s	0.4s	0.4s
	Server load	24.4%	24.4%	24.4%	12.2%	12.2%	12.2%
	Failed evictions	4316	1255	783	3562	1103	616
Read	Network requests	966460	837802	894124	829300	723878	734400
	Local index file(oram)	12K	12K	12K	13K	13K	13K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1914s	1574s	1696s	1809s	1511s	1552s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	24.4%	24.4%	24.4%	12.2%	12.2%	12.2%
	Failed evictions	9505	3067	1903	7293	2203	1160

Table 18: ORAM measurement result of Trivial ORAM with bucket size 1, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 2	**Bucket size**	1			2		
	Server capacity	16383			4094		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	380916	309960	326616	234360	239196	306912
	Local index file(oram)	6.5K	6.5K	6.5K	6.0K	6.0K	6.0K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1405s	759s	919s	398s	483s	610s
	time(File conversion)	1.4s	0.2s	0.5s	0.5s	0.2s	0.2s
	Server load	3.1%	3.1%	3.1%	12.2%	12.2%	12.2%
	Failed evictions	3027	760	378	895	143	56
Increment	Network requests	454356	356946	393576	274344	271560	355488
	Local index file(oram)	13K	13K	13K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1631s	1146s	1284s	468s	578s	703s
	time(File conversion)	1.2s	0.7s	0.9s	0.1s	0.6s	0.4s
	Server load	6.1%	6.1%	6.1%	24.4%	24.4%	24.4%
	Failed evictions	3707	951	558	1133	230	144
Read	Network requests	896076	748578	799056	559776	564324	750168
	Local index file(oram)	13K	13K	13K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	2913s	2385s	2572s	957s	1066s	1560s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	6.1%	6.1%	6.1%	24.4%	24.4%	24.4%
	Failed evictions	7297	2043	1148	2332	517	359

Table 19: ORAM measurement result of Trivial ORAM with bucket sizes 1 and 2, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 3	**Bucket size**	2					
	Server capacity	8190			16382		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	249504	269860	341880	274000	283404	382840
	Local index file(oram)	6.2K	6.2K	6.2K	6.4K	6.4K	6.4K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	456s	508s	675s	569s	645s	847s
	time(File conversion)	0.2s	0.2s	0.2s	0.2s	0.2s	0.3s
	Server load	6.1%	6.1%	6.1%	3.1%	3.1%	3.1%
	Failed evictions	856	155	55	870	127	63
Increment	Network requests	294584	294992	362208	326400	304648	397120
	Local index file(oram)	12K	12K	12K	13K	13K	13K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	547s	566s	703s	753s	802s	1032s
	time(File conversion)	0.4s	0.4s	0.4s	0.4s	0.4s	0.8s
	Server load	12.2%	12.2%	12.2%	6.1%	6.1%	6.1%
	Failed evictions	1101	216	88	1132	174	84
Read	Network requests	579416	593280	723184	627200	616076	792880
	Local index file(oram)	12K	12K	12K	13K	13K	13K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1066s	1132s	1422s	1506s	1547s	2044s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	12.2%	12.2%	12.2%	6.1%	6.1%	6.1%
	Failed evictions	2149	440	174	2136	363	166

Table 20: ORAM measurement result of Trivial ORAM with bucket size 2, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 4	**Bucket size**	4					
	Server capacity	4092			8188		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	242896	335984	488976	249312	376464	553104
	Local index file(oram)	5.9K	5.9K	5.9K	6K	6K	6K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	410s	602s	881s	443s	703s	1089s
	time(File conversion)	0.5s	0.2s	0.4s	0.9s	0.2s	0.2s
	Server load	12.2%	12.2%	12.2%	6.1%	6.1%	6.1%
	Failed evictions	229	6	1	242	6	1
Increment	Network requests	260224	349264	506544	294336	382416	555312
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	426s	656s	911s	519s	717s	1083s
	time(File conversion)	0.4s	0.4s	0.3s	0.4s	0.4s	0.4s
	Server load	24.4%	24.4%	24.4%	12.2%	12.2%	12.2%
	Failed evictions	356	26	19	376	14	3
Read	Network requests	526400	732392	1069696	598752	764088	1116144
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	917s	1328s	1922s	1116s	1428s	2227s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	24.4%	24.4%	24.4%	12.2%	12.2%	12.2%
	Failed evictions	850	103	96	782	27	11

Table 21: ORAM measurement result of Trivial ORAM with bucket size 4, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 5	**Bucket size**	4			8		
	Server capacity	16380			4088		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	289248	418592	616000	310080	587504	848000
	Local index file(oram)	6.2K	6.2K	6.2K	5.8K	5.8K	5.8K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	563s	931s	1332s	536s	1086s	1509s
	time(File conversion)	0.2s	0.8s	0.8s	0.2s	0.2s	0.2s
	Server load	3.1%	3.1%	3.1%	12.2%	12.2%	12.2%
	Failed evictions	286	8	0	70	3	0
Increment	Network requests	335616	426832	618464	359584	585168	853088
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	691s	998s	1403s	588s	1083s	1506s
	time(File conversion)	0.4s	0.9s	0.8s	0.4s	0.4s	0.4s
	Server load	6.1%	6.1%	6.1%	24.5%	24.5%	24.5%
	Failed evictions	412	18	2	161	1	3
Read	Network requests	641792	845424	1239392	703392	1193696	1745184
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1391s	1855s	2644s	1151s	2211s	3099s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	6.1%	6.1%	6.1%	24.5%	24.5%	24.5%
	Failed evictions	744	26	6	293	22	29

Table 22: ORAM measurement result of Trivial ORAM with bucket sizes 4 and 8, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 6	**Bucket size**	8					
	Server capacity	8184			16376		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	372704	664000	976000	395808	744000	1104000
	Local index file(oram)	5.9K	5.9K	5.9K	6K	6K	6K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	639s	1285s	2277s	762s	1535s	2258s
	time(File conversion)	0.2s	0.2s	1.8s	0.4s	1.2s	1.2s
	Server load	6.1%	6.1%	6.1%	3.1%	3.1%	3.1%
	Failed evictions	113	0	0	89	0	0
Increment	Network requests	395808	664000	976000	430080	744000	1104000
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	660s	1218s	2056s	863s	1582s	2341s
	time(File conversion)	0.9s	0.4s	3.0s	0.8s	1.6s	1.1s
	Server load	12.2%	12.2%	12.2%	6.1%	6.1%	6.1%
	Failed evictions	151	0	0	140	0	0
Read	Network requests	808032	1331984	1952000	885024	1489488	2208000
	Local index file(oram)	12K	12K	12K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1345s	2485s	3811s	1814s	3069s	4548s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	12.2%	12.2%	12.2%	6.1%	6.1%	6.1%
	Failed evictions	329	3	0	317	1	0

Table 23: ORAM measurement result of Trivial ORAM with bucket size 8, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 7	**Bucket size**	16					
	Server capacity	4080			8176		
	Eviction rate	1	3	5	1	3	5
Add	Network requests	506880	1008000	1440000	580992	1168000	1696000
	Local index file(oram)	5.6K	5.6K	5.6K	5.8K	5.8K	5.8K
	Local index file(file)	44K	44K	44K	44K	44K	44K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	927s	2025s	2612s	989s	2513s	3474s
	time(File conversion)	0.2s	1.6s	1.3s	0.4s	2.6s	2.0s
	Server load	12.3%	12.3%	12.3%	6.1%	6.1%	6.1%
	Failed evictions	28	0	0	34	0	0
Increment	Network requests	528960	1008000	1440000	598400	1168000	1696000
	Local index file(oram)	11K	11K	11K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	849s	1883s	2522s	1125s	2184s	3178s
	time(File conversion)	0.4s	3.1s	2.9s	1.2s	0.9s	0.5s
	Server load	24.5%	24.5%	24.5%	12.2%	12.2%	12.2%
	Failed evictions	51	0	0	50	0	0
Read	Network requests	1069440	2018016	2888640	1228352	2336000	3392000
	Local index file(oram)	11K	11K	11K	12K	12K	12K
	Local index file(file)	79K	79K	79K	79K	79K	79K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1741s	4044s	5982s	2787s	4639s	6603s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	24.5%	24.5%	24.5%	12.2%	12.2%	12.2%
	Failed evictions	114	1	3	129	0	0

Table 24: ORAM measurement result of Trivial ORAM with bucket size 16, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial - 8	**Bucket size**	16		
	Server capacity	16368		
	Eviction rate	1	3	5
Add	Network requests	615296	1328000	1952000
	Local index file(oram)	5.9K	5.9K	5.9K
	Local index file(file)	44K	44K	44K
	Shelter size	0	0	0
	time(ORAM)	1456s	3064s	4352s
	time(File conversion)	1.2s	1.2s	2.3s
	Server load	3.1%	3.1%	3.1%
	Failed evictions	6	0	0
Increment	Network requests	678528	1328000	1952000
	Local index file(oram)	12K	12K	12K
	Local index file(file)	79K	79K	79K
	Shelter size	0	0	0
	time(ORAM)	1607s	3045s	4328s
	time(File conversion)	2.3s	2.9s	3.8s
	Server load	6.1%	6.1%	6.1%
	Failed evictions	58	0	0
Read	Network requests	1397184	2656000	3904000
	Local index file(oram)	12K	12K	12K
	Local index file(file)	79K	79K	79K
	Shelter size	0	0	0
	time(ORAM)	3220s	6100s	8836s
	time(File conversion)	0.1s	0.1s	0.1s
	Server load	6.1%	6.1%	6.1%
	Failed evictions	149	0	0

Table 25: ORAM measurement result of Trivial ORAM with bucket size 16, the init step is not displayed because all measurements are initiated with 0% basic load.

Trivial 25% load - 1	**Bucket size**	2			4		
	Server capacity	8190			4092		
	Eviction rate	1	3	5	1	3	5
Init	Network requests	1213112	1207984	1536304	513152	698528	1026752
	Local index file(oram)	27K	27K	27K	12K	12K	12K
	Local index file(file)	182K	182K	182K	91K	91K	91K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	2342s	2409s	3007s	854s	1279s	1854s
	time(File conversion)	1.8s	1.5s	1.6s	0.4s	0.6s	0.5s
	Server load	25%	25%	25%	25%	25%	25%
	Failed evictions	4546	885	447	665	29	29
Add	Network requests	385848	362148	458920	274816	413672	613904
	Local index file(oram)	33K	33K	33K	18K	18K	18K
	Local index file(file)	226K	226K	226K	135K	135K	135K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	722s	697s	897s	450s	731s	1098s
	time(File conversion)	1.1s	0.8s	0.7s	0.9s	0.6s	0.5s
	Server load	31.1%	31.1%	31.1%	37.2%	37.2%	37.2%
	Failed evictions	1597	379	245	404	123	129
Increment	Network requests	403328	430128	596288	394288	725752	1316624
	Local index file(oram)	40K	40K	40K	25K	25K	25K
	Local index file(file)	261K	261K	261K	170K	170K	170K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	760s	858s	1165s	641s	1270s	2381s
	time(File conversion)	1.2s	0.9s	1.3s	0.6s	0.9s	0.6s
	Server load	37.2%	37.2%	37.2%	49.4%	49.4%	49.4%
	Failed evictions	1692	544	468	797	593	849
Read	Network requests	819352	884564	1235080	843296	1985360	4244624
	Local index file(oram)	40K	40K	40K	25K	25K	25K
	Local index file(file)	261K	261K	261K	170K	170K	170K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1158s	1757s	2424s	1372s	3490s	7527s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	37.2%	37.2%	37.2%	49.4%	49.4%	49.4%
	Failed evictions	3453	1147	1005	1774	1990	3349

Table 26: ORAM measurement result of Trivial ORAM with bucket sizes 2 and 4, all measurements are initiated with 25% basic load.

Trivial 25% load - 2	**Bucket size**	4					
	Server capacity	8188			16380		
	Eviction rate	1	3	5	1	3	5
Init	Network requests	1201536	1583976	2321712	2744176	3580280	5184256
	Local index file(oram)	25K	25K	25K	54K	54K	54K
	Local index file(file)	182K	182K	182K	363K	363K	363K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	2087s	2940s	4400s	5633s	7201s	10487s
	time(File conversion)	1.6s	1.3s	1.1s	20.3s	22.9s	27.6s
	Server load	25%	25%	25%	25%	25%	25%
	Failed evictions	1529	82	56	3362	250	113
Add	Network requests	324576	444168	653568	350704	458968	697312
	Local index file(oram)	32K	32K	32K	61K	61K	61K
	Local index file(file)	226K	226K	226K	408K	408K	408K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	563s	818s	1232s	664s	943s	1433s
	time(File conversion)	0.7s	0.7s	0.7s	1.7s	1.2s	2s
	Server load	31.1%	31.1%	31.1%	28.1%	28.1%	28.1%
	Failed evictions	466	97	92	453	57	66
Increment	Network requests	365568	479136	821376	365424	458968	762608
	Local index file(oram)	38K	38K	38K	68K	68K	68K
	Local index file(file)	261K	261K	261K	442K	442K	442K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	630s	884s	1550s	729s	961s	1538s
	time(File conversion)	0.8s	0.8s	0.8s	1.4s	0.7s	2.2s
	Server load	37.2%	37.2%	37.2%	31.1%	31.1%	31.1%
	Failed evictions	588	144	244	493	84	119
Read	Network requests	684096	1060200	1668144	730848	1008576	1557248
	Local index file(oram)	38K	38K	38K	68K	68K	68K
	Local index file(file)	261K	261K	261K	552K	552K	552K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1190s	1945s	3139s	1390s	2012s	3150s
	time(File conversion)	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
	Server load	37.2%	37.2%	37.2%	31.1%	31.1%	31.1%
	Failed evictions	1036	425	511	986	224	264

Table 27: ORAM measurement result of Trivial ORAM with bucket size 4, all measurements are initiated with 25% basic load.

Trivial 25% load - 3	**Bucket size**	8		
	Server capacity	8184		
	Eviction rate	1	3	5
Init	Network requests	1595392	2723728	4001600
	Local index file(oram)	25K	25K	25K
	Local index file(file)	181K	181K	181K
	Shelter size	0	0	0
	time(ORAM)	2716s	5759s	7671s
	time(File conversion)	1.4s	9.8s	9.9s
	Server load	25%	25%	25%
	Failed evictions	578	5	4
Add	Network requests	421344	714464	1052128
	Local index file(oram)	32K	32K	32K
	Local index file(file)	226K	226K	226K
	Shelter size	0	0	0
	time(ORAM)	708s	1802s	1917s
	time(File conversion)	0.7s	5.2s	3.9s
	Server load	31.1%	31.1%	31.1%
	Failed evictions	193	38	39
Increment	Network requests	411008	756960	1167296
	Local index file(oram)	38K	38K	38K
	Local index file(file)	260K	260K	260K
	Shelter size	0	0	0
	time(ORAM)	684s	1611s	2116s
	time(File conversion)	0.8s	12.6s	2.3s
	Server load	37.2%	37.2%	37.2%
	Failed evictions	176	70	98
Read	Network requests	843296	1689216	2551264
	Local index file(oram)	38K	38K	38K
	Local index file(file)	260K	260K	260K
	Shelter size	0	0	0
	time(ORAM)	1412s	3341s	4675s
	time(File conversion)	0.1s	0.1s	0.1s
	Server load	37.2%	37.2%	37.2%
	Failed evictions	387	272	307

Table 28: ORAM measurement result of Trivial ORAM with bucket size 8, all measurements are initiated with 25% basic load.

Trivial 50% load - 1	**Bucket size**	2			4		
	Server capacity	8190			4092		
	Eviction rate	1	2	3	1	2	3
Init	Network requests	3292496	3163424	3608708	1225120	1384336	1883768
	Local index file(oram)	54K	54K	54K	25K	25K	25K
	Local index file(file)	363K	363K	363K	181K	181K	181K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	6061s	6450s	7122s	2039s	2399s	3334s
	time(File conversion)	4.2s	20s	6.3s	1.7s	1.3s	1.8s
	Server load	50%	50%	50%	50%	50%	50%
	Failed evictions	13799	6311	4664	1984	745	791
Add	Network requests	1490952	1466192	2454284	1112336	—	—
	Local index file(oram)	61K	61K	61K	32K		
	Local index file(file)	408K	408K	408K	226K		
	Shelter size	0	0	0	0		
	time(ORAM)	2745s	2968s	4651s	1791s		
	time(File conversion)	1.2s	2.4s	1.1s	0.7s		
	Server load	56.1%	56.1%	56.1%	62.2%		
	Failed evictions	7603	4323	5457	3159		
Increment	Network requests	—	—	—	—	—	—
	Local index file(oram)						
	Local index file(file)						
	Shelter size						
	time(ORAM)						
	time(File conversion)						
	Server load						
	Failed evictions						
Read	Network requests	—	—	—	—	—	—
	Local index file(oram)						
	Local index file(file)						
	Shelter size						
	time(ORAM)						
	time(File conversion)						
	Server load						
	Failed evictions						

Table 29: ORAM measurement result of Trivial ORAM with bucket sizes 2 and 4, all measurements are initiated with 50% basic load. Note that cells marked as '—' indicates that the measurement test has been aborted before or during this step. This is because Trivial ORAM measurements with a high load lead to incredibly slow measurements after reaching a certain standard.

Trivial 50% load - 2	**Bucket size**	4					
	Server capacity	8188			16380		
	Eviction rate	1	2	3	1	2	3
Init	Network requests	2800560	3120456	4350912	6426016	6981056	9548512
	Local index file(oram)	52K	52K	52K	109K	109K	109K
	Local index file(file)	363K	363K	363K	727K	727K	727K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	4949s	5844s	7986s	13374s	15047s	20693s
	time(File conversion)	5.1s	4.1s	3.5s	46.8s	43.8s	48.6s
	Server load	50%	50%	50%	50%	50%	50%
	Failed evictions	4241	1559	1754	9272	3292	3398
Add	Network requests	699888	1155336	14946216	769856	1042112	2193488
	Local index file(oram)	58K	58K	58K	116K	116K	116K
	Local index file(file)	408K	408K	408K	771K	771K	771K
	Shelter size	0	0	0	0	0	0
	time(ORAM)	1253s	2055s	27146s	1598s	2211s	4482s
	time(File conversion)	1.3s	1.2s	1.1s	9.9s	11.8s	19.5s
	Server load	56.1%	56.1%	56.1%	53.1%	53.1%	53.1%
	Failed evictions	1583	1593	19589	1592	1214	2162
Increment	Network requests	2531088	—	—	1064624	1717600	18081856
	Local index file(oram)	65K			123K	123K	123K
	Local index file(file)	442K			806K	806K	806K
	Shelter size	0			0	0	0
	time(ORAM)	4539s			2167s	3612	37502s
	time(File conversion)	2s			15.6s	17.1s	22.3s
	Server load	62.2%			56.1%	56.1%	56.1%
	Failed evictions	7033			2393	2325	21444
Read	Network requests	17276112	—	—	1993456	3563488	—
	Local index file(oram)	65K			123K	123K	
	Local index file(file)	442K			806K	806K	
	Shelter size	0			0	0	
	time(ORAM)	30287s			3825s	7417s	
	time(File conversion)	0.1s			0.1s	0.1s	
	Server load	62.2%			56.1%	56.1%	
	Failed evictions	50417			4417	4861	

Table 30: ORAM measurement result of Trivial ORAM with bucket size 4, all measurements are initiated with 50% basic load. Note that cells marked as '—' indicates that the measurement test has been aborted before or during this step. This is because Trivial ORAM measurements with a high load lead to incredibly slow measurements after reaching a certain standard.

Trivial 50% load - 3	**Bucket size**	8		
	Server capacity	8184		
	Eviction rate	1	2	3
Init	Network requests	3424256	4761600	6880368
	Local index file(oram)	51K	51K	51K
	Local index file(file)	363K	363K	363K
	Shelter size	0	0	0
	time(ORAM)	5632s	8577s	13056s
	time(File conversion)	3.8s	3.7s	9.1s
	Server load	50%	50%	50%
	Failed evictions	1540	708	1089
Add	Network requests	649952	3586080	—
	Local index file(oram)	58K	58K	
	Local index file(file)	407K	407K	
	Shelter size	0	0	
	time(ORAM)	1048s	6336s	
	time(File conversion)	1.4s	1.2s	
	Server load	56.1%	56.1%	
	Failed evictions	569	3115	
Increment	Network requests	1335776	—	—
	Local index file(oram)	64K		
	Local index file(file)	442K		
	Shelter size	0		
	time(ORAM)	2149s		
	time(File conversion)	1.3s		
	Server load	62.2%		
	Failed evictions	1697		
Read	Network requests	5562592	—	—
	Local index file(oram)	64K		
	Local index file(file)	442K		
	Shelter size	0		
	time(ORAM)	8961s		
	time(File conversion)	0.1s		
	Server load	62.2%		
	Failed evictions	8149		

Table 31: ORAM measurement result of Trivial ORAM with bucket size 8, all measurements are initiated with 50% basic load. Note that cells marked as '—' indicates that the measurement test has been aborted before or during this step. This is because Trivial ORAM measurements with a high load lead to incredibly slow measurements after reaching a certain standard.