UNIVERSITY OF THE AEGEAN
SCHOOL OF ENGINEERING

DEPARTMENT OF INFORMATION AND COMMUNICATION SYSTEMS ENGINEERING

POSTGRADUATE STUDIES PROGRAMME

MSc IN INFORMATION AND COMMUNICATION SYSTEMS SECURITY

IMPLEMENTATION OF OBLIVIOUS DATA STRUCTURES

POSTGRADUATE THESIS

by

Orestis Th. Anavaloglou

Supervisor: Assistant Professor Dr. Panagiotis Rizomiliotis

Members of examination committee: Prof. Dr. S. Kokolakis, Assoc. Prof. Dr. M. Karyda

Samos, September 2020

Acknowledgements

First of all, I would like to thank my wife, Katerina and my children, Fanis, Irini and Christina for their patience and support throughout this endeavor. The Academic and Scientific achievements of my close friends, Dr. Stelios Choulis and Dr. Ifigenia Klaoudatou, have always been an inspiration for me. I deeply thank them for that, but I mostly thank them for being my friends all these years. I also want to thank my sister, Electra Anavaloglou, for the time she spent checking and discussing every linguistic detail of the text with me. Her degree in English literature has proven to be invaluable. A truly inspirational person for me for many years has been Dr. Dimos Charmpis. I treasure our long conversations and I deeply thank him for his support. I definitely feel indebted to many of my teachers since my school days, but I would like to express my deepest gratitude to my supervisor, Dr. Panagiotis Rizomiliotis, for his insight and precious help whenever I needed them. Lastly, I would like to thank my parents for all their sacrifices. I know I cannot ever repay them, but I promise to do the same for my children when the time comes. After all, every generation asks for nothing more from the next one.

© 2020
by
ORESTIS THEOFANIS ANAVALOGLOU

Department of Information and Communication Systems Engineering
UNIVERSITY OF THE AEGEAN

Contents

Abstract

We present the key historical advances in Cryptography that led to the concept of Oblivious RAM (ORAM) by Goldreich and Ostrovsky in 1993. We briefly present their first algorithmic construct, Hierarchical ORAM, followed by a very different approach, the Binary-tree based ORAM by Shi et al. (2011) and its improvement for Multi-Party Computation (MPC), Circuit ORAM. We thoroughly examine Path-ORAM algorithm by Stefanov et al. (2012) and we present our Python implementation of it. We move on to the definition of an Oblivious Data Structure (ODS) and the description of the pointer-based technique to handle Oblivious Data Structures proposed by Shi et al. along with their ODS framework. We examine the architecture of the framework and we present our python implementation. We finally present three basic ODS algorithms and their Python counterparts. Namely, Oblivious Stack, Oblivious Queue and Oblivious Heap (priority queue).

Keywords: *Cryptography, ORAM, Oblivious Data Structures, ODS framework, Python, Oblivious Stack, Oblivious Queue and Oblivious Heap (priority queue).*

1

Introduction

1.1 *A brief history of Cryptography*

One of the core structural elements of human society that can be traced back to the beginnings of writing, is "secrets". Spanning from ancient Egypt and Greece to our technologically advanced era, secrecy has proven to be a necessity for all the central civilizations of this planet through the ages. People need to protect all kinds of messages, all the time, in almost every aspect of their lives. The knowledge of a sole secret strategic move of the opponent can be decisive for a battle, even for a whole war. In politics, in diplomacy, in economic life, even in personal affairs, there is a definite and continuous need to conceal and safeguard vast amounts of information.

Essentially, two methods are selectively implemented in order to achieve the protection of a secret piece of information. Namely, Cryptography and Steganography. The latter could be defined as the practice of concealing a secret message inside a publicly accessible carrier-message in such a way that there is no evidence for the mere existence of the secret one. A good example of this technique is the embedding of a message in a jpeg image by properly altering some of its LSB's (Least Significant Bits). The resulting image is practically the same with the initial one thus the hidden message is untraceable. Although the historical and practical importance of Steganography is not a matter of debate, its innate weakness - the fact that once the hidden message has been discovered, it is simultaneously also recovered by the attacker – makes Cryptography stand out as the dominant tool to rely on for keeping humanity's secrets safe and sound.

One of the earliest examples of a form of encryption would be the **hieratic writing** of the priests in ancient Egypt which was considered sacred and was kept secret from the laymen under

the orders of the Pharaohs. Plutarch, in his biography of Lysander, the Spartan admiral, gives the description of a subsequent cryptographic method supposedly applied at the city-state of Sparta, in ancient Greece in the 5th century B.C. It was called the **Skytale** and was used to safely convey messages between the military and the city officials, the five "ephors". In its simplest form, a strip of cloth or parchment was wrapped around a cylindrical wooden stick and the message was written on it. Then the strip was unwound rendering the message unintelligible. To recover the message, the recipient had only to rewrap the strip around a cylinder of exactly the same radius with the one used in the first place.

Among the several simplistic methods of encryption in antiquity, probably the most advanced was the one used by Julius Caesar in ancient Rome. The "Caesar cipher", as it is called, consists in substituting each letter of the message with the letter sitting a fixed number of places to its left or right in the alphabet. Julius Caesar, for example, usually used a right shift of three places. This meant that 'A' would be replaced by 'D', 'B' would be replaced by 'E' and so on. Although such a monoalphabetic substitution cipher was considered unbreakable for a thousand years, this notion could not be further from the truth. During the Middle Ages, the Arab world was flourishing in the fields of Science and Mathematics. In such a favorable environment, an Arab scholar discovered that each letter of an alphabet has a certain frequency of appearance in any given text. This led to the development of a cryptanalytic method, called "frequency analysis", which resulted in breaking a substitution cipher with comparatively small effort.

The evolution from monoalphabetic ciphers to polyalphabetic ones came from Leon Battista Alberti and his treatise *De Cifris* of 1467 where he describes the design and use of his cipher disk. A device consisting of two concentric rings with the outer ring being stationary holding an alphabet for plaintext and the inner one being movable holding a garbled alphabet for ciphertext. In mid-sixteenth century, Italian cryptologist Giovan Battista Bellaso described a polyalphabetic substitution cipher that added a new element in the world of Cryptography. Misattributed to French diplomat, cryptographer and alchemist Blaise de Vigenère (1523–1596), the now known as the Vigenère cipher introduced the concept of the **encryption key**. This exceptional cipher for the history of Cryptography was finally broken by the German infantry officer Friedrich Kasiski in 1863.

In 1882, Frank Miller described a system for securing telegraphy. A secret message could be encrypted by adding respectively each letter of the message with the corresponding letter of a sequence (at least equal to the message in length) of random letters using modular addition (mod 26). In 1917 his idea was further improved by Gilbert Vernam who two years later was granted a U.S. patent for the XOR operation used in his method of encryption now called "Vernam cipher" or "One-time pad". In the 1940's, exceptional mathematician and information theorist Claude Shannon proved that the One-time pad system actually provided what is known as "perfect secrecy". This means that if the key (pad) used to encrypt the message has the following characteristics:

1) it is truly random,
2) its length is at least equal to that of the plaintext message,

3) it is never reused in whole or partially,
4) it is successfully kept completely secret,

then the ciphertext produced by the one-time pad system will be impossible to decrypt. Despite that these four prerequisites were serious drawbacks and a wide adoption of the system was quite unlikely, the high level of security it provides has given One-time pad a special place in the world of Cryptography. A quite indicative fact is that the NSA used One-time pad tapes even until the '70s while the modern stream ciphers essentially emulate the One-time pad scheme by XORing the plaintext with their keystream to produce the ciphertext.

Around the end of WWI, German engineer Arthur Scherbius invented a cryptographic rotor machine which produced the ciphertext while the plaintext was being typed on its keyboard. Its name was "Enigma" and it was massively used by Nazi Germany during WWII. The initial cryptanalysis efforts to break it were carried out by French and Polish cryptanalysts while in Bletchley Park, England, a team of codebreakers (including Alan Turing) developed the technology to decrypt the daily keys which in Enigma's case meant the initial state of its rotors. The monumental achievements of the *Government Code and Cypher School* (GC&CS) in Bletchley Park contributed enormously to the victory of the Allies and the end of the Second World War but they also marked the beginning of our computer era through the invention of *Colossus*, the world's first programmable digital computer.

The incredibly fast pace at which computer technology advanced in the coming years paved the way to the advent of the modern cryptographic primitives such as the one-way hash functions, the stream and block ciphers as well as a whole new concept of encryption: Public-key cryptography, which also provided the base for Digital Signature schemes, another immensely important development. On the other hand, the increasing dependence of our modern technological society on computer systems as well as the milestones conquered concerning the available computational power and the inevitable roll-out of such powerful hardware to consumers world-wide, introduced a variety of new threats and a completely new level of potential damages that malevolent third parties could inflict. One of the most serious problems that arose in the last few decades is **software piracy** which has evolved into a scourge for the software developers and the companies operating in the software production sector. Another, more recent issue, appeared due to the fact that much of today's computer related work is done through the Internet one way or the other and a significant percentage of the data produced are stored in servers across the "cloud". So, it was necessary to develop tools in order to protect all kinds of sensitive data from potentially untrusted servers. As strange as it may seem, an idea proposed for software protection, also offered a solution for the "untrusted server" problem.

2

Oblivious RAM (ORAM)

2.1 *Overview*

Oded Goldreich introduced the idea of *Oblivious* RAM (or "ORAM") in the late '80s. In 1993 he co-authored a paper [6] with Rafail Ostrovsky in an effort to establish theoretically the **software protection** problem. They set their starting point by concluding that it is impossible to ensure that a person with a legitimate copy of a program would not be able to produce executable copies of it, given that only software measures of prevention are taken. The reason is that any computer software (even if it is encrypted) is nothing more than a sequence of bits that can be copied, one by one, resulting in an executable copy of the whole program. Since therefore the hardware **has** to play a role as well, seems plausible to examine whether a hardware-only approach would suffice. But this would lead to the manufacturing of different computer systems, designed specifically for each application in the market which would be, of course, insanely impractical. In other words, the solution should be a mixture of software and hardware. For example, a physically shielded (secure) CPU holding a cryptographic key, combined with an encrypted program. This Software-Hardware-package (SH-package) could be installed by connecting the secure CPU to the user's computer and loading the program in its encrypted form to its memory. During runtime, the secure CPU would be charged with the following tasks:

- Decrypt the program's instructions and execute them.
- Encrypt everything the program writes to computer's RAM.
- Decrypt everything the program reads from RAM.

Even though the SH-package approach described above is certainly in the correct direction, there is still a serious issue it is not addressing. Since the memory locations of the sequence of RAM calls are not concealed, the whole access pattern of the RAM ("memory access pattern") is observable and exploitable. Of course, this is catastrophic because the information it leaks (e.g. the loop structure of the code) can lead even to the complete reconstruction of the encrypted program.

In this context, Goldreich and Ostrovsky proposed the idea of **Oblivious RAM** which they defined as a probabilistic RAM for which the probability distribution of the sequence of memory addresses accessed during an execution depends only on the running time (i.e. it is independent of the particular input). Alternatively, this means that **the sequence of memory accesses reveals no information about the input (to the ORAM) beyond the running-time for this specific input.**

This inspiring idea turned out to be extremely useful when cloud computing begun being widely used and the need to fully protect large amounts of data residing on outsourced (untrusted by default) servers became clear and imperative. It is sufficient to just perform the thought experiment of replacing the **secure CPU** and **RAM** components of the RAM model with a **client** and an **untrusted server** respectively and it immediately becomes evident that the problem of leaking information through the data access patterns emerges, equally severe, in both cases.

2.2 *The evolution of ORAM*

Ostrovsky's proposed solution called ***Hierarchical ORAM*** achieved an overhead of $O(log^3 N)$ through continuous shuffling and re-encrypting of the sensitive data and many of the ORAM algorithms that have been developed since, are based on it. In a nutshell, the algorithm allocates memory for $\boldsymbol{N = (1 + \lceil log_4 t \rceil)}$ "buffers" where the ith level buffer is a hash table containing 4^i "buckets". Each bucket is of size $m = O(logt)$, where $\boldsymbol{t}$ is equal to the input length. Apparently, the Nth level buffer will contain $4^N = 4^{(1+\lceil log_4 t \rceil)} = 4 \cdot 4^{\lceil log_4 t \rceil} = 4t$ buckets (Fig. 1). For each level, a random s_i and a hash function $h_{s_i}(\cdot)$ associated with it are chosen and kept secret. During a *read* operation, one data block of each buffer level is read even if the requested one may have been found before reaching the last Nth level. It is essential for the obliviousness of the whole procedure to keep on reading *dummy* blocks until N of them are read from the ORAM. The *write* operation is always performed on the 1st level buffer; thus, it is expected to become full quite fast. So, in order to avoid overflows, the contents of buffer level **i** are moved to buffer level **i+1** (every 4^{i-1} retrieves) where they are obliviously hashed together with the pre-existing contents of this level. A high-level description of the Hierarchical ORAM algorithm would be the following:

```
Read/Write block a
01: Scan all buckets in level 1 for block a
02: If found, store block a locally and continue with dummy block b as input
03: For i = 2 to N:
04:    Compute h_{s_i}(a) [or h_{s_i}(b) if block a is already found]
05:    Read the corresponding bucket
06:    Scan the bucket for block a
07:    If found, store a locally and continue with dummy block b as input
08:    Write the bucket back
09: (Write) Modify the data in block a if necessary
10: Write block a back on the first level
```

Hierarchical data structure

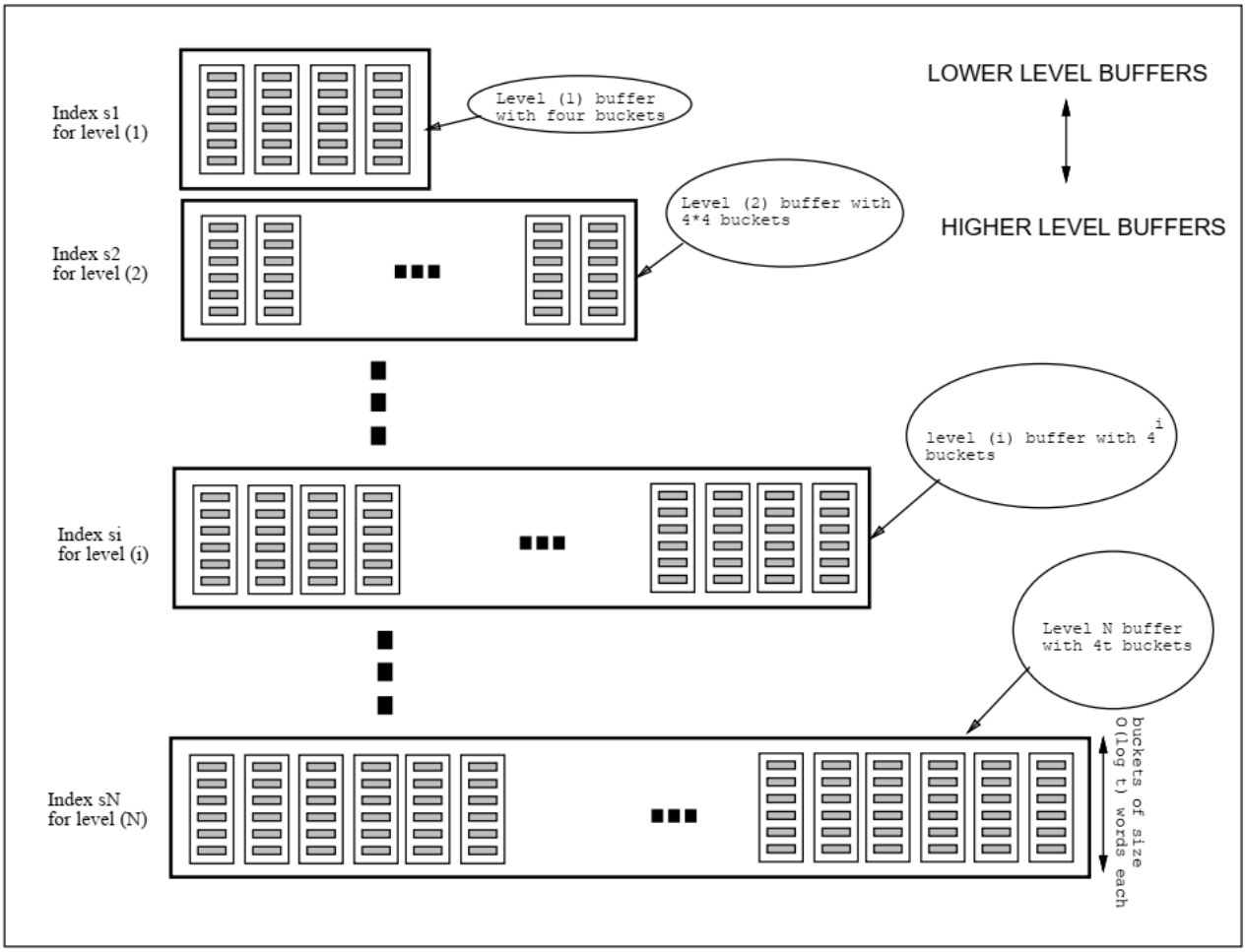


Fig. 1 [6]

A totally different approach to ORAM was proposed by Shi E., Chan T.H.H., Stefanov E., Li M. [7] in 2011. The ***Binary-tree based ORAM*** maintains the same amortized cost ($O(log^3 N)$ using Trivial Bucket scheme) but achieves a much lower worst-case cost of $O(log^3 N)$ compared to the $O(Nlog^2 N)$ of the Hierarchical ORAM. In their construction, the server storage is organized into a binary tree whose nodes are small "buckets" with a fixed number of slots each. A slot can contain either a real data block or a "dummy" one. Every data block is randomly assigned to a leaf node of the binary tree and performing a *read* operation translates into reading the whole path from the root to this leaf node (Fig. 2). If a block is to be written back in the ORAM, it is first assigned to a new random path and then added to the root bucket to avoid linkability between the two paths given that the root node belongs to every possible path of the binary-tree. As expected, an "eviction" method is also included to prevent overflows of the root bucket through percolating the data blocks towards the leaves of the tree. In order to keep track of the path that is assigned to each block, a relatively small position map is stored on the client. Applying the ORAM construction recursively over this position map eventually leads to $O(1)$ client-side storage.

Binary-tree based ORAM

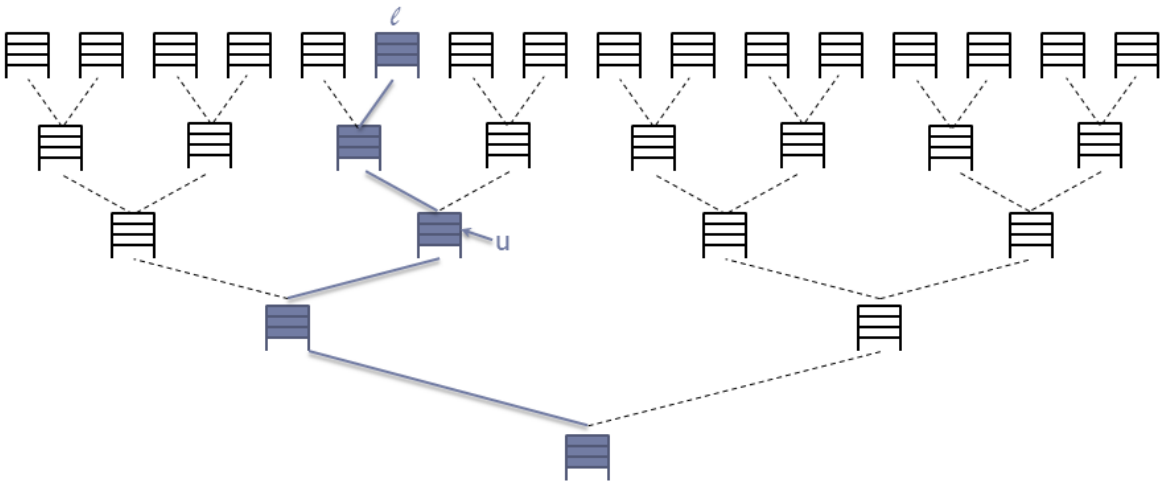


Fig. 2 [7]

Building further upon the idea of tree based ORAM and the improvements of its performance for multi-party computation (MPC) achieved by Gentry et al. [8], Wang, Chang and Shi came up with the concept of ***Circuit ORAM*** [9]. An ORAM's *circuit complexity* is the total circuit size of the ORAM client algorithm over all rounds of interaction and as for several MPC protocols, **XOR** operations are essentially free, the primary performance metric for the MPC case should be the number of **AND** gates. The aim of Circuit ORAM algorithm is to improve the eviction method (requiring $logN$ data block scans in previous tree-based ORAMs) through **two metadata scans** (to compute all the information necessary for the client to develop foresight) and **only one scan of the data blocks** on the eviction path from the stash to the leaf. A very informative table (found in [9]) comparing several ORAM schemes with respect to their asymptotic and concrete circuit size is the following :

Scheme	Circuit Size (asymptotic)*	# AND gates (concrete)**
Hierarchical ORAM [6]	$O(Dlog^3N + C_{PRF}log^2N)$	$\geq$ 476.1M
Binary-tree ORAM [7]	$O\big((D + log^2N)log^2N\big)\omega(1)$	30.1M
Path ORAM (naive circuit) [10]	$O\big((D + log^2N)log^2N\big)\omega(1)$	56.6M
Path ORAM (o-sort circuit) [11]	$O\big((D + log^2N)logNloglogN\big)\omega(1)$	41.4M
Circuit ORAM [9]	$\boldsymbol{O\left((D + log^2N)logN\right)\omega(1)}$	**0.97M**

*: The variable C_{PRF} denotes the circuit size of a PRF function with input size of $O(logN)$ bits.

**: The concrete circuit size is calculated based on 4GB data with a 32-bit block size, with 2^{-80} security failure probability.

2.3 *The Path ORAM algorithm*

A real breakthrough on *Oblivious RAMs* came from Emil Stefanov et al. in 2012 [10] when they presented their Path ORAM algorithm. An extremely simple and efficient approach that requires only a small client storage and achieves $O(logN)$ overhead for data blocks of size $B = \Omega(log^2 N)$ bits. The key idea is to treat the storage space (e.g. the untrusted server) as a binary tree where each one of its nodes (called a *bucket*) contains a fixed number of encrypted data blocks. On the other side, the client maintains a local data structure (called the *stash*) where it keeps a small number of the data blocks. Also, at any given moment, each block is mapped to a uniformly random leaf node of the binary tree. These (block, leaf) tuples form the *position map* that is also kept on the client side. Thus, if a block **a** is mapped to position **x**, it means that **a** resides either in a *bucket* along the path from the **root** to the **xth** leaf node of the ORAM's binary tree or in the *stash*.

N	Total # blocks outsourced to server
$L = \lceil \log_2 N \rceil - 1$	Height of binary tree
B	Block size (in bits)
Z	Capacity of each bucket (in blocks)
$\mathcal{P}(x)$	path from leaf node x to the root
$\mathcal{P}(x, \ell)$	the bucket at level ℓ along the path $\mathcal{P}(x)$
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 $\mathcal{P}(x)$ or in the stash.

Access(op, a, data*):

1: $x \leftarrow$ position[a]
2: position[a] $\leftarrow$ UniformRandom$(0 \ldots 2^L - 1)$
3: **for** $\ell \in \{0, 1, \ldots, L\}$ **do**
4: $\quad S \leftarrow S \cup$ ReadBucket$(\mathcal{P}(x, \ell))$
5: **end for**
6: data $\leftarrow$ Read block a from S
7: **if** op $=$ write **then**
8: $\quad S \leftarrow (S - \{(\mathsf{a}, \mathsf{data})\}) \cup \{(\mathsf{a}, \mathsf{data}^*)\}$
9: **end if**
10: **for** $\ell \in \{L, L-1, \ldots, 0\}$ **do**
11: $\quad S' \leftarrow \{(\mathsf{a}', \mathsf{data}') \in S : \mathcal{P}(x, \ell) = \mathcal{P}(\mathsf{position}[\mathsf{a}'], \ell)\}$
12: $\quad S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from S'.
13: $\quad S \leftarrow S - S'$
14: $\quad$ WriteBucket$(\mathcal{P}(x, \ell), S')$
15: **end for**
16: **return** data

Fig. 3 [10]

The *read* and *write* operations are implemented using a single protocol called *Access* (Fig. 3) which can be concisely described as follows:

Lines 1-2: **Remap block** → Assign a new random position to block **a** while **x** holds its old position.

Lines 3-5: **Read path** → Read the path $P(x)$ that contains block **a**.

Lines 6-9: **Update block** → If the user performs a *write* operation, update the data in block **a**.

Lines 10-15: **Write path** → Write the path back to the ORAM, starting from the leaf and including blocks from the stash whose paths intersect at a certain tree level with the one being written. In other words, satisfying the condition: $P(x, l) = P(position[a'], l)$.

Our python implementation of Path ORAM as a function called ***oramAccess*** (excluding some lines of code for clarity that can be found anyway in the Appendix) is shown below:

```
def oramAccess(op, block_node):
    global S
    S = []                                                  # Initialize local stash as a list of tuples
    oramPath = []

    if op != 'readandremove' and op != 'add': raise ValueError

    jnode = json.dumps(block_node.__dict__)                     # Serialize object block_node to JSON
    dnode = json.loads(jnode)                                   # Turn JSON into python dictionary

    x = dnode['pos']
    oramPath = oram.P(oram.nod[L, x])               # Get path of leaf x and store it locally in a list

    # Read the block in question from the local stash
    block = next((a for a in S if a[0] == dnode['label']), ('None', 'Null', 0, {}))
    if op == 'add':                     # If the operation is 'add':
        print('(Add)')
        if block in S:
            S.remove(block)             # Remove the old block from the stash if it's there
        # Add the new block, data and its children positions or the old block with new data
        S.append((dnode['label'], dnode['data'], dnode['pos'], dnode['chPos']))

    if block in S:
        S.remove(block)
    S_temp = []
    for l in range(L, -1, -1):
```

```
        # S_temp = {b in S : P(x, l) = P(position[b], l)}
        S_temp = [b for b in S if oram.Pl(oram.nod[L, x], l) == oram.Pl(oram.nod[L, b[2]], l)])]
        # S_temp = {Select min(|S_temp|, Z) elements from S_temp}
        S_temp = S_temp[:min(len(S_temp), Z
        # S = S - S_temp
        S = [item for item in S if item not in S_temp]
        # WriteBucket(P(x, l), S_temp)
        writeBucket(oram.Pl(oram.nod[L, x], l), S_temp)

    if op == 'readandremove':
        print('(ReadAndRemove)')
        askedBlock = odnode.Odnode(block[0], block[1], block[2], json.loads(str(block[3]).replace("'", '"')
))
        return askedBlock
```

One of the key choices we made concerning the whole project is the data type to represent the concept of a **node** in the oblivious data structures. Given the fact that this data type should be able to hold several data fields of various types themselves, we came up with the following python class, called **Odnode** (**O**blivious **d**ata **node**):

```
class Odnode:
    def __init__(self, label, data, pos, chPos):
        self.label = label
        self.data = data
        self.pos = pos
        self.chPos = chPos
```

So, every **Odnode** object holds the node's:

- id → **label** (type: string)
- payload → **data** (type: string)
- position tag in the ORAM → **pos** (type: integer)
- children's id's and positions in the ORAM → **chPos** (type: dictionary)

The reason why the **Odnode** class contains the fields **pos** and **chPos** that are not directly connected with the functionality of Path ORAM will become evident in the next chapter where the ODS framework is discussed.

3

Oblivious Data Structures

3.1 *Overview*

A ***data structure*** can be generally defined as a specialized format for organizing, processing, retrieving and storing data. The important role *data structures* play in Computer Science is undoubtedly proclaimed in the title of the book *"Algorithms + Data Structures = Programs"*, written by Turing Award winner Niklaus Emil Wirth. The idea behind the ***oblivious*** *data structures* is that it should be possible to create more efficient constructions that achieve lower asymptotic blowup than the generic ORAM schemes because the algorithms of building and managing common data structures exhibit a certain predictability in their access patterns. The access pattern of general RAM programs can be a complete graph since they make arbitrary random accesses to data but common data structures have a sparser access pattern graph than generic RAM programs because there are some obvious limitations depending on the type of the structure. For example, memory accesses on a binary search tree can only go from one tree node to an adjacent one. This observation led to the assumption that there is an efficiency margin to be gained compared to ORAMs by not hiding some publicly known aspects of the common data structures' access patterns.

Consequently, in their 2014 paper [12], Elaine Shi, Emil Stefanov et al. presented the following definition of an **Oblivious Data Structure (ODS)**:

Definition (Oblivious data structure). *We say that a data structure D is oblivious if there exists a polynomial-time simulator S, such that for any polynomial-length sequence of data structure operations $\overrightarrow{ops} = \left((op_1, arg_1), \ldots, (op_M, arg_M)\right)$*

$$addresses_D(\overrightarrow{ops}) \equiv S\left(L(\overrightarrow{ops})\right)$$

where $addresses_D(\overrightarrow{ops})$ is the physical addresses generated by the oblivious data structure during a sequence of operations $\overrightarrow{ops}$; and $L(\overrightarrow{ops})$ is referred to as the leakage function. Typically, we consider that $L(\overrightarrow{ops}) = M$, i.e., the number of operations is leaked, but nothing else.

This definition implies that a data structure is *oblivious* if the access pattern produced by a sequence of operations on the data structure reveals **only** the total number of these operations and nothing else. In other words, no polynomial-time distinguisher can distinguish the sequence of real addresses generated by the oblivious data structure from a sequence of addresses produced by a polynomial-time simulator S with knowledge of only the total number of operations.

Two different techniques to create and handle oblivious data structures are described in the paper depending on the type of sparse access pattern graph each data structure generates:

- ***Locality-based***, for access pattern graphs with low doubling dimensions (out of the scope of this thesis).
- ***Pointer-based***, for access pattern graphs that are rooted trees with bounded degree which can achieve $O(logN)$ (Fig. 4) bandwidth blowup, a significant improvement compared to best known ORAM.

Technique	Example Applications	Client-side storage	Blowup
Pointer-based for rooted tree access pattern graph	`map/set`, `priority_queue`, `stack`, `queue`, oblivious memory allocator	$O(\log N) \cdot \omega(1)$	$O(\log N)$
Locality-based for access pattern graph with doubling dimension dim	maximum flow, random walk on sparse graphs; shortest-path distance on planar graphs; `doubly-linked list`, `deque`	$O(1)^{\text{dim}} \cdot O(\log^2 N) + O(\log N) \cdot \omega(1)$	$O(12^{\text{dim}} \log^{2-\frac{1}{\text{dim}}} N)$
Path ORAM [45]	All of the above	$O(\log N) \cdot \omega(1)$	$O(\frac{\log^2 N + \chi \log N}{\chi})$ for block size $\chi \log N$
ORAM in [29]	All of the above	$O(1)$	$O(\frac{\log^2 N}{\log \log N})$

Fig. 4 [12]

3.2 *The ODS framework*

Building further upon the core concept of *oblivious data structures*, Shi's team came up with a practical construction of their approach. A generalized framework where the nodes of the data structure are stored (in encrypted form) in a non-recursive, position-based ORAM on the server. Due to its overall very good performance and simplicity, Path ORAM was their scheme of choice. Each **node** of the data structure has the following format:

$$\mathbf{node} \coloneqq (\mathbf{id}, \mathbf{data}, \mathbf{pos}, \mathbf{childrenPos})$$

Where **data** is the payload of the node, **id** is its identifier and **pos**, its position tag in the ORAM. What makes this construction special though, is the **childrenPos** field. This is a mapping from every child node's id to its position tag. Keeping this information in every node means that the need of a position map on the client side is obsolete. It is obviously sufficient to just store the **id** and **pos**

of the root node to be able to actually traverse the whole data structure as long as its access pattern graph is a rooted tree of bounded degree.

The framework communicates with the underlying ORAM through two basic functions:

- **ReadAndRemove**(**id**, **pos**): fetches the node identified by **id** which is at position **pos** in the ORAM and removes it from the server.

- **Add**(**id**, **pos**, **data**): writes the node identified by **id**, which contains the payload **data** to some location among a set of locations indicated by **pos**.

Once a node is fetched from the server through a *ReadAndRemove* call, its position tag is revealed to the server. So, there is a need for a new position tag to be generated for this node. Of course, this means the position tag in its parent's children position list should also be updated.

The functionality of the framework relies on a relatively small client-side cache of $O(logN)$ size that stores all the nodes which are read and removed from the server when they are needed for an operation. This way the client is able to perform whatever updates are requested **locally** and more importantly without fetching any nodes twice from the server before the operation is complete and they are written back. The aforementioned *updates* can be insertions, removals as well as modifications of graph structures. When these have finished, the cache nodes are assigned new uniformly random position tags, their parent nodes are updated in order to point to the correct positions of their children, they are written back to the server and finally, the cache is emptied.

An important detail that, if not taken care of, can lead to information leakage is that every cache miss during an operation will generate requests to the server. Thus, the number of accesses to the server should always be the same independently of the operation being performed on the data structure. Therefore, the operations are padded with dummy *ReadAndRemove* and *Add* calls to the maximum number required by any data structure operation.

3.3 *ODS framework architecture*

The oblivious data structures algorithms are implemented through calls to the ODS client. Its Access protocol supports four types of operations:

1. **(Read, id)** → Read the node identified by **id**
2. **(Write, id, data*)** → Write the node **id** holding the payload **data***
3. **(Insert, id, data*)** → Insert a new node **id** with payload **data***
4. **(Del, id)** → Delete the node **id**

Assuming the untrusted server supports two basic functions, such as **GET(physical_address)** and **PUT(physical_address, data)**, a high-level overview of the framework's architecture (Fig. 5) can be described as follows:

1) One single call to **ODS.Start** in order to prepare for the operation (fetch the root node).

2) A sequence of **ODS.Access** calls in order to read and remove from the server the nodes for the requested operation and perform all the necessary updates to them locally (in cache). It is worth noting that not every ODS.Access call will result to a *ReadAndRemove* operation, since nodes in cache can be directly returned. However, as mentioned before, this cannot lead to information leakage thanks to the padding performed by the ODS client on the next stage which ensures that every operation results in the same number of *ReadAndRemove* and *Add* calls to the ORAM.

3) One single call to **ODS.Finalize** in order to:

   a) Assign new random positions (for the ORAM) to all the nodes in cache and update their parents' **childrenPos** entries to match them.

   b) Write all the nodes in cache back to the server through a sequence of *Add* calls to the ORAM.

   c) Perform the necessary padding to ensure that every data structure operation generates the same number of *ReadAndRemove* and *Add* calls to the underlying position-based ORAM.

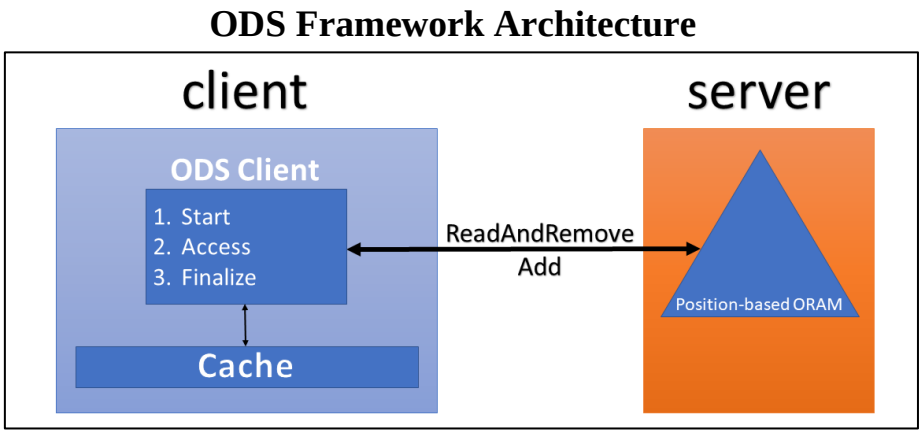


Fig. 5

ODS Framework Algorithm

```
rootPos:  contains the root of the data structure
data:  contains the data_node and childrenIDs
childrenPos:  map from id to pos for all children of a node

ODS.Start():
 1: Update cache to contain rootPos
ODS.Access(op, id, data*):
 1: if op = Insert then
 2:     cache.insert(id, data*, null, null) // No pos, childrenPos yet.  Updated during Finalize()
 3: else if op = Read then
 4:     if cache.contains(id) then
 5:         (id, data, pos, childrenPos) := cache.get(id)
 6:     else
 7:         pos := get pos from cache using childrenPos entries
 8:         (id, data, pos, childrenPos) := ReadAndRemove(id, pos)
 9:         cache.insert(id, data, pos, childrenPos)
10:     end if
11:     return data
12: else if op = Write then
13:     if cache.contains(id) then
14:         cache.update(id, data*) // pos, childrenPos remain the same
15:     else
16:         pos := get pos from cache using childrenPos entries
17:         (id, data, pos, childrenPos) := ReadAndRemove(id, pos)
18:         cache.insert(id, data, pos, childrenPos)
19:     end if
20: else if op = Del then
21:     if cache.contains(id) then
22:         cache.remove(id)
23:     else
24:         pos := get pos from cache using childrenPos entries
25:         ReadAndRemove(id, pos)
26:     end if
27: end if
ODS.Finalize(rootID, padVal):
 1: Generate UniformRandom pos for all nodes in cache
 2: Update childrenPos of all nodes based on their id and new pos
 3: Update rootPos based on the new rootID
 4: Pad ReadAndRemove() to padVal
 5: while not cache.empty() do
 6:     (id, data, pos*, childrenPos*) := cache.deleteNextElement()
 7:     Add(id, data, pos*, childrenPos*)
 8: end while
 9: Pad Add() to padVal
```

Fig. 6 [12]

The pseudo code of the ODS client for dynamic data structures published by Shi et al. can be seen in Fig. 6 while our python implementation is the following:

```python
##############################   ODS Framework Functions   ##############################


def odsStart():                                     # Update cache to contain the root
    global cache
    global root
    cache.clear()
    if root != None:
        cache.append(root)
```

```
###################  Read  ###################

def read(nodeLabel):
    global cache

    isInCache = any(x.label == nodeLabel for x in cache)        # True if the block is already in cache
    if isInCache == False:
        n = 0
        while isInCache == False:                                           #
            childDictKeys = list(cache[n].chPos.keys())                     #
            childName = childDictKeys[0]                                    #
            childPosition = cache[n].chPos[childName]                       # Traverse through the ..
            if not any(x.label == childName for x in cache):                # .. nodes using their ..
                ask = odnode.Odnode(childName, 'null', childPosition, {})   # .. children positions ..
                fetch = oramAccess('readandremove', ask)                    # .. until the ..
                cache.append(fetch)                                         # .. requested one ..
            isInCache = any(x.label == nodeLabel for x in cache)            # .. is found.
            n += 1                                                          #
    return cache[-1]                                         # Return the last object (node) in cache


####################  Insert  ####################

def insert(newNodeLabel, newNodeData):
    global cache
    global root
    global top

    newNode = odnode.Odnode(newNodeLabel, newNodeData, 0, {})   # Create Odnode instance for the new block
    if root != None: oramAccess('readandremove', root)          # Remove root from ORAM
    cache.insert(0, newNode)                                    # Insert new block in cache at index 0
    root = newNode                                              # newNode is the new root
    if top == None:                                             # If the structure was empty, newNode is..
        top = newNode                                           # .. also the new top.

#################   Update(Write)   ################

def write(nodeLabel, newData):
    global cache
    isInCache = any(x.label == nodeLabel for x in cache)           # True if the block is already in cache
    if isInCache == False:
        read(nodeLabel)
    next((n for n in cache if n.label == nodeLabel)).data = newData


#################   Delete   #################

def delete(nodeLabel):
    global cache
```

```
    global root

    if len(cache) == 0:
        print('\nThe Oblivious Data Structure is empty!\n')

    else:
        if root != None: oramAccess('readandremove', root)                # Remove root from ORAM
        read(nodeLabel)                                                   # Get the node from ORAM
        del cache[-1]                                                     # Delete from cache


################   Finalize   ################

def finalize(typeIs):
    print('finalize()')
    global cache
    global root
    global top

    if cache != [] and typeIs != 'enqueue':
        # Assign new random position to each node in cache
        for n in cache:
            pos = random.randint(0, 2**L - 1)
            n.pos = pos

        # Update children's positions in each node
        if typeIs == 'linear':
            for i, j in enumerate(cache):
                if i < len(cache)-1:
                    cName = cache[i+1].label         # Assign to cName current block's child label
                    cPos = cache[i+1].pos            # Assign to cPos current block's child position
                    j.chPos = {cName : cPos}         # Add to current block the pair {Child_id : position}
                    if i == 0:                       # Store the root of the ..
                        root = j                     # .. data structure in variable 'root'


        if typeIs == 'heap':
            cacheNodeDict = dict((x.label, x.pos) for x in cache)

            for i, j in enumerate(cache):
                if i < len(cache)-1:
                    childrenList = list(j.chPos.keys())

                    if len(childrenList) > 0:
                        if childrenList[0] in list(cacheNodeDict.keys()):
                            j.chPos[childrenList[0]] = cacheNodeDict[childrenList[0]]


                    if len(childrenList) > 1:
```

```
                    if childrenList[1] in list(cacheNodeDict.keys()):
                        j.chPos[childrenList[1]] = cacheNodeDict[childrenList[1]]

                if i == 0:                                  # Store the root of the ..
                    root = j                                # .. data structure in variable 'root'

# Write cahe back to ORAM
for k in cache:
    oramAccess('add', k)

# Empty client cache
cache.clear()
```

Although it may seem strange that the *finalize* function takes the type of the data structure as an argument, it will become evident in the next chapter that it is due to certain features of the oblivious data structures algorithms that this was actually necessary.

4

Python Implementation

4.1 *Oblivious Stack*

As it can be seen in Fig. 7, the pseudocode of the *Oblivious Stack* algorithm consists of the two basic functions of a stack: *Push(data$_{node}$)* and *Pop()*. In ***Push***, variable **id** is assigned a new value by the function **nextid()** and the new node is pushed to the stack by calling **ODS.Access** and inserting it at the top of the stack. Before calling **ODS.Finalize**, the variable **top** is updated to point to the new top node of the stack. ***Pop*** just reads the top node of the stack (so, it is removed from the ORAM) and keeps it locally. Then deletes it from cache and assigns to the pointer **top** the next node in the stack.

```
Push(data_node):
1: ODS.Start()
2: id := nextid()
3: ODS.Access(Insert, id, (data_node, top))
4: top := id
5: ODS.Finalize(top, padVal = 1)
Pop():
1: ODS.Start()
2: (data_node, nextid) := ODS.Access(Read, top, null)
3: ODS.Access(Del, top, null)
4: top := nextid
5: ODS.Finalize(top, padVal = 1)
6: return data_node
```

Fig. 7 [12]

In our application, we have implemented two functions that appear in all the data structures' option menus and are not specific to this one. These are:

- **IsEmpty()**: checks if the oblivious structure is empty. Once the *odsStart()* is called at the beginning of every operation, the root node is expected to reside in cache. So, the only case where the size of the cache equals zero (len(cache) == 0 → True) is when the oblivious structure is actually empty.

- **Path ORAM explorer**: displays the contents of the ORAM's binary tree on the server.

The section of our program referring to the Oblivious Stack is the following:

```
        print('Oblivious Stack Options')
        print('_______________________')
        print('[1] --> Push(item)')
        print('[2] --> Pop()')
        print('[3] --> IsEmpty()')
        print('---------------------------')
        print('[4] --> Path ORAM explorer')
        print('[ENTER] --> EXIT')
        print('_____________________________')

        select = input('Please enter your choice : ')

        if select == '':
            break

        odsStart()
        if select == '1':
            newBlockName = input('\nEnter the ID of the item you want to push : ')
            newBlockData = input("Enter the data of item '{0}' : ".format(newBlockName))
            print()

            def push(node, data):
                insert(node, data)
                finalize('linear')

            push(newBlockName, newBlockData)

            print('\nOperation finished successfully!')
            input('\nPlease press [ENTER] to continue...')

        if select == '2':
            print()
```

```
def pop():
    global cache
    global root

    if root != None:
        oldTop = oramAccess('readandremove', root)

        if oldTop.chPos == {}:                              # If this is the last item
            newRoot = None
        else:
            rootChildKey = list(oldTop.chPos.keys())[0]     # Get the root's child label
            newRoot = read(rootChildKey)                    # Read root's next item
            del cache[0]                                    # Delete old root (top)
        root = newRoot
        finalize('linear')
    else:
        oldTop = None

    return oldTop

topItem = pop()

if topItem != None:
    print('\nItem ID :', topItem.label)
    print('Item Data :', topItem.data)

else:
    print('\nThe Oblivious Stack is empty!')

input('\nPlease press [ENTER] to continue...')
```

When the options menu is displayed, the user can select between ***Push(item)*** (insert a new data block in the stack) and ***Pop()*** (display the top element of the stack and delete it from the data structure). Their functionality is described below:

- **Push(item)**: The new block's label and data are provided by the user and then the function *push(node, data)* is called which simply consists of calling *insert(node, data)* and then *finalize('linear')*.

- **Pop()**: The top node (root) is assigned to variable oldTop and removed from the stack. Then the next node is fetched by calling the function *read(rootChildKey)* where variable *rootChildKey* holds the label (id) of the child of the old top node. This node becomes the new top of the oblivious stack before the *finalize()* is called. Of course, certain precautions are taken in case the stack is already empty or it has only one last element. The node oldTop is returned by the function and its label and data are displayed to the user.

4.2 *Oblivious Queue*

In Oblivious Queue, the algorithm needs to maintain two pointers (**head** and **tail**) instead of one (**top**) which was the case in Oblivious Stack. The pseudocode as presented in the original paper can be seen below (Fig. 8).

head/tail: head/tail of the queue

Enqueue($data_{node}$):

1: ODS.Start()
2: $head^* := nextid()$
3: ODS.Access(Insert, head, ($data_{node}$, $head^*$))
4: $head := head^*$
5: ODS.Finalize(tail, padVal = 1)

Dequeue():

1: ODS.Start()
2: ($data_{node}$, $tail^*$) := ODS.Access(Read, tail, null)
3: ODS.Access(Del, tail, null)
4: $tail := tail^*$
5: ODS.Finalize(tail, padVal = 1)
6: **return** $data_{node}$

Fig. 8 [12]

Our Oblivious Queue implementation is the following:

```
        print('Oblivious Queue Options')
        print('_________________________')
        print('[1] --> Enqueue(item)')
        print('[2] --> Dequeue()')
        print('[3] --> IsEmpty()')
        print('--------------------------')
        print('[4] --> Path ORAM explorer')
        print('[ENTER] --> EXIT')
        print('_____________________________')

        select = input('Please enter your choice : ')

        if select == '':
            break

        odsStart()
        if select == '1':
            global nextID
            global nextPOS
            global queueSize

            newID = nextID
```

```
            newPOS = nextPOS

            queueSize += 1
            nextID = str(int(newID) + 1)
            nextPOS = random.randint(0, 2**L - 1)       # Generate extra random pos for next enqueue()

            newBlockData = input("\nEnter the data of item '{0}' : ".format(newID))
            print()
            newNode = odnode.Odnode(newID, newBlockData, newPOS, {nextID : nextPOS})

            def enqueue(qnode):
                global root
                global queueSize

                cache.clear()
                cache.append(qnode)
                finalize('enqueue')
                if queueSize == 1:
                    root = newNode

            enqueue(newNode)

            print('\nOperation finished successfully!')
            input('\nPlease press [ENTER] to continue...')


        if select == '2':
            print()

            def dequeue():
                global cache
                global root
                global queueSize

                if root != None:
                    oldHead = oramAccess('readandremove', root)
                    queueSize -= 1
                    rootChildKey = list(oldHead.chPos.keys())[0]

                    if queueSize == 0:                          # If this is the last item
                        newRoot = None
                    else:
                        newRoot = read(rootChildKey)            # Read root's next item
                        del cache[0]                            # Delete old root (top)
                    root = newRoot
                    finalize('linear')
                else:
                    oldHead = None
```

```
            return oldHead

        headItem = dequeue()

        if headItem != None:
            print('\nItem ID :', headItem.label )
            print('Item Data :', headItem.data)

        else:
            print('\nThe Oblivious Queue is empty!')

        input('\nPlease press [ENTER] to continue...')


    if select == '3':

        def isQueueEmpty():
            return (len(cache) == 0)

        ans = isQueueEmpty()

        if ans:
            print('\nTRUE - The Oblivious Queue is empty.')
        else:
            print('\nFALSE - The Oblivious Queue is NOT empty.')

        input('\nPlease press [ENTER] to continue...')
```

From this options menu the user can select between ***Enqueue(item)*** (insert a new data block at the tail of the oblivious queue) and ***Dequeue()*** (display the head element of the queue and remove it from the data structure). Their functionality is described below:

- **Enqueue(item)**: The new block's label (id) is provided by **nextID** (which equals the id of the tail, incremented by 1) while its data come from user's input. In order to be able to add new elements at the tail without the obligation to update the previous tail's child position tag (and thus necessarily traverse the whole queue), three essential operations take place:
    1. Increment **nextID** by 1.
    2. Generate a new random position tag for **nextPOS**, and
    3. Incorporate these features in the newNode object's child position field in order to prepare it for the next possible Enqueue().

- **Dequeue()**: The head node (root) is assigned to variable oldHead and removed from the oblivious queue. Then the next node in the queue is fetched by calling the function *read(rootChildKey)* and it is subsequently assigned to the head pointer (newRoot) before

finalize() is called. Of course, as in the Oblivious Stack section, certain precautions are again taken in case the queue is already empty or it has only one last element. The node oldHead is returned by the function and its id and data are displayed to the user.

4.3 *Oblivious Heap (Priority Queue)*

The algorithm for Oblivious Heap (Fig. 9 and Fig. 10) consists of three functions:

1. **ReadPath()**: Reads and removes from the ORAM the path from the root of the heap to the leaf that is its last node. Every node on this path is appended to a list which is eventually returned by the function.

2. **Insert(key)**: After the *ODS.Start()* and *ReadPath()* have been executed, the new key is appended to the list returned by *ReadPath()*. Then the list is traversed once by a "for" loop in order to restore the **min-heap property** wherever it has been violated. Finally, the nodes are moved to the local cache by calling *ODS.Access(Write)* and then the *ODS.Finalize()* call writes the whole path (together with the new node) back to the ORAM.

3. **ExtractMin()**: Exactly as in *Insert(key)* above, *ODS.Start()* and *ReadPath()* are executed at the beginning of the whole process. As the structure is a **min-heap** the node carrying the minimum key resides at the root of the tree. Therefore, it is the first node in the list returned by *ReadPath()*. So, it is assigned to variable **key*** and then it is removed from the structure and replaced by the **last node** of the list returned by *ReadPath()* (which happens to be also the last node of the whole heap). Then the tree is traversed starting from the new root and going down a path detecting and restoring the min-heap property throughout the heap. In the end, the value of **key*** is returned by the function.

An important difference that should be noticed between the algorithms of the two "linear" oblivious structures (stack and queue) and this binary tree structure is that in the former case, we have a padding value of 1 ($padVal = 1$) while in the latter case of the oblivious heap $padVal = 3logN$.

Oblivious Heap (Priority Queue)

```
ReadPath():
 1: nodeid := 1 level := 1 list := ()
 2: while nodeid ≤ size do
 3:     (key) := ODS.Access(Read, nodeid, null)
 4:     append (key, nodeid) at the end of list
 5:     if level-th least significant bit in size is 0 then
 6:         nodeid := nodeid ∗ 2
 7:     else
 8:         nodeid := nodeid ∗ 2 + 1
 9:     end if
10:     level := level + 1
11: end while
12: return list
Insert(key):
 1: ODS.Start()
 2: list := ReadPath()
 3: append (key, size) at the end of list
 4: size := size + 1
 5: for i := length(list) downto 2 do
 6:     if list[i].key > list[i − 1].key then
 7:         swap(list[i].key, list[i − 1].key)
 8:     end if
 9: end for
10: for i : length(list) downto 1 do
11:     ODS.Access(Write, list[i].nodeid, list[i].key)
12: end for
13: ODS.Finalize(1, padVal = 3 log N)
```

Fig. 9 [12]

```
ExtractMin():
 1: ODS.Start()
 2: list := ReadPath()
 3: key* := list[1].key
 4: list[1].key := list[length(list)].key
 5: size := size − 1
 6: for i : length(list) − 1 downto 1 do
 7:     ODS.Access(Write, list[i].nodeid, list[i].key)
 8: end for
 9: nodeid := 1
10: while nodeid ≤ size do
11:     key := ODS.Access(Read, nodeid, null)
12:     lkey := ODS.Access(Read, 2nodeid, null)
13:     rkey := ODS.Access(Read, 2nodeid + 1, null)
14:     if rkey < lkey then
15:         if key ≥ rkey then
16:             ODS.Access(Write, nodeid, rkey)
17:             nodeid := nodeid ∗ 2 + 1
18:         end if
19:     else if lkey ≤ rkey then
20:         if key ≥ lkey then
21:             ODS.Access(Write, nodeid, lkey)
22:             nodeid := nodeid ∗ 2
23:         end if
24:     end if
25:     ODS.Access(Write, nodeid, key)
26: end while
27: ODS.Finalize(1, padVal = 3 log N)
28: return key*
```

Fig. 10 [12]

In our *ReadPath()* python implementation we had to distinguish between the call from *ExtractMin()* and the call from *Insert(key).* This had to be done due to the fact that in our heap, the id's of the nodes are given by the user and are NOT automatically attributed by the program as an integer sequence starting from 0 or 1. This has the consequence that when *Insert(key)* is called, *ReadPath()* must return, as last node of the path, the last candidate **parent** node for the new node to be attached to, and not the actual last node of the heap that is needed for the *ExtractMin()* algorithm. For example, if the last node of the tree is a leaf at the rightmost position of a certain level l, then the new node must be attached to the **leftmost** node of this level in order to start level $l + 1$. Another difference is that our function does not return a list but draws the path directly into the cache instead:

```
def readPath(operation):
    global cache
    global root
    global last
    depth = math.floor(math.log2(last))
    currentNode = oramAccess('readandremove', root)
    binLast = last + 1

    if ((binLast & (binLast - 1)) != 0) or (operation == 'extract'):       # Last node IS NOT at the end of a tree level ..
        for k in range(depth-1, 0, -1):                                    # .. or the call is from extractMin()
                # Check if the last parent has already 2 children. If yes, go to the next
                if k == 1 and (last % 2 == 1) and (operation == 'insert'):
                    ind = math.floor(last/2)
                else:
                    ind = math.floor(last/math.pow(2,k)) - 1
                if (ind % 2) == 1:
                    leftChildLabel = list(currentNode.chPos.keys())[0]
                    leftChildPos = currentNode.chPos[leftChildLabel]
                    ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})     # Ask for the left child ..
                    fetch = oramAccess('readandremove', ask)                          # .. to be fetched from ORAM
                else:
                    rightChildLabel = list(currentNode.chPos.keys())[1]
                    rightChildPos = currentNode.chPos[rightChildLabel]
                    ask = odnode.Odnode(rightChildLabel, 'null', rightChildPos, {})   # Ask for the right child ..
                    fetch = oramAccess('readandremove', ask)                          # .. to be fetched from ORAM

                cache.append(fetch)                                                   # Append fetched node to cache
                currentNode = fetch
        else:                                                                         # Last node IS at the end of a level
            for k in range(depth):
                leftChildLabel = list(currentNode.chPos.keys())[0]
                leftChildPos = currentNode.chPos[leftChildLabel]
                ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})           # Ask for the left child ..
                fetch = oramAccess('readandremove', ask)                                # .. to be fetched from ORAM
                cache.append(fetch)                                                     # Append fetched node to cache
                currentNode = fetch
```

Our python version of *Insert()* function, as it can be seen below, is called *insertKey* and takes the **id** and the **key** of the new node as arguments. Firstly, it creates the new *Odnode* instance with the given arguments and appends it to the end of the cache after *readPath()* has been called. Then the "Upheap" process starts which is responsible for restoring the min-heap property while at the same time updates the children nodes mappings (**chPos** dictionaries' keys) wherever is necessary.

```
def insertKey(id, key):
            global cache
            global root
            global last

            newNode = odnode.Odnode(id, key, 0, {})                                    # Create Odnode instance for the new node

            if last == 0:
                root = newNode
                oramAccess('add', newNode)
                last = 1
            else:
                readPath('insert')
                cache.append(newNode)                                                  # Append new node in cache
                cache[-2].chPos[cache[-1].label] = cache[-1].pos                       # Attach new node to the heap

                ################################  Upheap  ################################
                k = len(cache)-1
                while (k > 0) and (int(cache[k].data) < int(cache[k-1].data)):
                    cache[k-1].label, cache[k].label = cache[k].label, cache[k-1].label        # Swap cache objects id's to restore order
                    cache[k-1].data, cache[k].data = cache[k].data, cache[k-1].data             # Swap cache objects keys to restore order

                    childKeys = list(cache[k-1].chPos.keys())

                    ########################################################################## If swapped node was left child
                    if childKeys[0] == cache[k-1].label:
                        if len(childKeys) == 2:
                            newPos = {cache[k-1].label : cache[k].label, childKeys[1] : childKeys[1]}
                            cache[k-1].chPos = dict((newPos[key], value) for (key, value) in cache[k-1].chPos.items())
                        else:
                            newPos = {cache[k-1].label : cache[k].label}
                            cache[k-1].chPos = dict((newPos[key], value) for (key, value) in cache[k-1].chPos.items())

                        # If swapped node hasn't reached the root
                        if k-2 >= 0:
                            childKeysParent = list(cache[k-2].chPos.keys())

                            # If swapped node was left child
                            if childKeysParent[0] == cache[k].label:
                                newPosParent = {cache[k].label : cache[k-1].label, childKeysParent[1] : childKeysParent[1]}
                                cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())
```

```
                # If swapped node was right child
                if childKeysParent[1] == cache[k].label:
                    newPosParent = {childKeysParent[0] : childKeysParent[0], cache[k].label : cache[k-1].label}
                    cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

        ######################################################################## If swapped node was right child
        if len(childKeys) > 1 and childKeys[1] == cache[k-1].label:
            newPos = {childKeys[0] : childKeys[0], cache[k-1].label : cache[k].label}
            cache[k-1].chPos = dict((newPos[key], value) for (key, value) in cache[k-1].chPos.items())

            # If swapped node hasn't reached the root
            if k-2 >= 0:
                childKeysParent = list(cache[k-2].chPos.keys())

                # If swapped node was left child
                if childKeysParent[0] == cache[k].label:
                    newPosParent = {cache[k].label : cache[k-1].label, childKeysParent[1] : childKeysParent[1]}
                    cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

                # If swapped node was right child
                if childKeysParent[1] == cache[k].label:
                    newPosParent = {childKeysParent[0] : childKeysParent[0], cache[k].label : cache[k-1].label}
                    cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

        k -= 1

    last += 1
    finalize('heap')
```

Finally, our *extractMin()* python function executes first a *readPath()* call and fills the local cache with the nodes belonging to the path from the root to the parent of the last node of the heap. It assigns the root node (which obviously holds the minimum value) to the variable **min** and then fetches the last node of the heap from the ORAM and replaces the root with it. Of course, the last node in cache must be deleted and the heap's size must be reduced by 1. Then the "Downheap" process starts, aiming to put the new root to the right place in the tree preserving the min-heap property. During "Downheap", if a parent node has a key with greater value than the smaller of the two (or one) children nodes, their labels (id's) and data (keys) are swapped and the algorithm continues the comparison downwards until the "new root" reaches a node where the min-heap property is not violated. Of course, when a swap takes place, the children nodes mappings (the keys in chPos dictionaries showing **which** are the children nodes) must be updated on the current node (parent) and on the previous one (parent's parent) except when the current node in comparison is the root and there is no previous one. The python code of *extractMin()* is the following:

```
def extractMin():
            global cache
            global root
            global last

            if last > 0:
                readPath('extract')
                currentNode = cache[-1]

                # If last node in cache is not a leaf, fetch another one
                if currentNode.chPos != {}:
                    ind = last - 1

                    if (ind % 2) == 1:
                        leftChildLabel = list(currentNode.chPos.keys())[0]
                        leftChildPos = currentNode.chPos[leftChildLabel]
                        ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})       # Ask for the left child ..
                        fetch = oramAccess('readandremove', ask)                            # .. to be fetched from ORAM
                    else:
                        rightChildLabel = list(currentNode.chPos.keys())[1]
                        rightChildPos = currentNode.chPos[rightChildLabel]
                        ask = odnode.Odnode(rightChildLabel, 'null', rightChildPos, {})     # Ask for the right child ..
                        fetch = oramAccess('readandremove', ask)                            # .. to be fetched from ORAM

                    cache.append(fetch)                                                     # Append fetched node to cache

                min = (cache[0].label, cache[0].data)                                       # Assign minimum element to min
                cache[0].label = cache[-1].label                                            # Last element becomes the new ..
                cache[0].data  = cache[-1].data                                             # .. root leaving chPos's as they are
                del cache[-1]                                                               # Remove last element from cache

                if len(cache) > 0:
                    del cache[-1].chPos[cache[0].label]                  # Remove previous last element from its parent's chPos dictionary

                last -= 1

                ################################  Downheap  ################################

                if last > 0:
                    currentNode = cache[0]
                    previousNode = None

                    k = 0
                    while k <= math.floor(last/2) - 1:
                        childKeys = list(currentNode.chPos.keys())
```

```
if len(childKeys) > 0:                                          # If current node has at least 1 child
    leftChildLabel = childKeys[0]
    leftChildPos = currentNode.chPos[leftChildLabel]

    isInCache = any(x.label == leftChildLabel for x in cache)       # True if the node in question is already in cache
    if isInCache == False:
        ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})           # Ask for the left child ..
        leftChild = oramAccess('readandremove', ask)                            # .. to be fetched from ORAM
        cache.append(leftChild)                                                 # Add left child to cache
    else:
        leftChild = next(n for n in cache if n.label == leftChildLabel)
    indexLeft = next((i for i, item in enumerate(cache) if item.label == leftChildLabel), -1)

    rightChild = None                                                           # Initialize right child node

if len(childKeys) > 1:                                                          # If current node has 2 children
    rightChildLabel = childKeys[1]
    rightChildPos = currentNode.chPos[rightChildLabel]

    isInCache = any(x.label == rightChildLabel for x in cache)      # True if the node in question is already in cache
    if isInCache == False:
        ask = odnode.Odnode(rightChildLabel, 'null', rightChildPos, {})         # Ask for the right child ..
        rightChild = oramAccess('readandremove', ask)                           # .. to be fetched from ORAM
        cache.append(rightChild)                                                # Add right child to cache
    else:
        rightChild = next(n for n in cache if n.label == rightChildLabel)
    indexRight = next((i for i, item in enumerate(cache) if item.label == rightChildLabel), -1)

if len(childKeys) == 0:
    # The node has no children
    break

if rightChild != None:
    if int(leftChild.data) < int(rightChild.data):
        if int(currentNode.data) > int(leftChild.data):
            # Swap cache objects id's to restore order
            currentNode.label, cache[indexLeft].label = cache[indexLeft].label, currentNode.label
            # Swap cache objects keys to restore order
            currentNode.data, cache[indexLeft].data = cache[indexLeft].data, currentNode.data

            # Correction of the chPos dictionary (label:data) pairs in current node
            if len(childKeys) == 2:
                newPos = {currentNode.label : cache[indexLeft].label, childKeys[1] : childKeys[1]}
                currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())
            else:
                newPos = {currentNode.label : cache[indexLeft].label}
                currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())
```

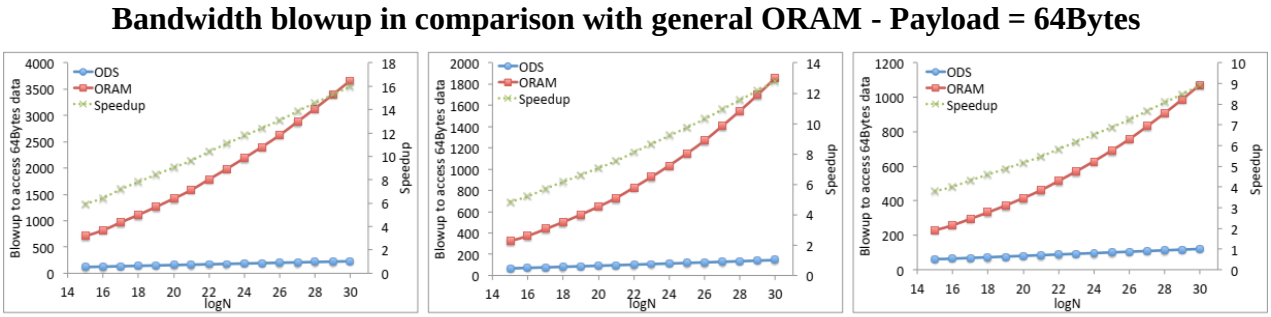
```
                    # Correction of the chPos dictionary (label:data) pairs in previous node
                    if previousNode != None:
                        previousChildKeys = list(previousNode.chPos.keys())

                        if cache[indexLeft].label == previousChildKeys[0]:                # If swapped node was a left child
                            newPos = {cache[indexLeft].label : currentNode.label, previousChildKeys[1] : previousChildKeys[1]}
                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())
                        else:                                                             # If swapped node was a right child
                            newPos = {previousChildKeys[0] : previousChildKeys[0], cache[indexLeft].label : currentNode.label}
                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())

                    previousNode = currentNode                                          # Assign current node to previousNode
                    currentNode = cache[indexLeft]                                      # Let current node be the left child
                    k = 2*k + 1                                                         # Move node index to left child

                else:
                    # The root node reached the right position in the heap
                    break

            else:
                if int(currentNode.data) > int(rightChild.data):
                    # Swap cache objects id's to restore order
                    currentNode.label, cache[indexRight].label = cache[indexRight].label, currentNode.label
                    # Swap cache objects keys to restore order
                    currentNode.data, cache[indexRight].data = cache[indexRight].data, currentNode.data

                    # Correction of the chPos dictionary (label:data) pairs in current node
                    newPos = {childKeys[0] : childKeys[0], currentNode.label : cache[indexRight].label}
                    currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())

                    # Correction of the chPos dictionary (label:data) pairs in previous node
                    if previousNode != None:
                        previousChildKeys = list(previousNode.chPos.keys())

                        if cache[indexRight].label == previousChildKeys[0]:               # If swapped node was a left child
                            newPos = {cache[indexRight].label : currentNode.label, previousChildKeys[1] : previousChildKeys[1]}
                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())
                        else:                                                             # If swapped node was a right child
                            newPos = {previousChildKeys[0] : previousChildKeys[0], cache[indexRight].label : currentNode.label}
                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())

                    previousNode = currentNode                                          # Assign current node to previousNode
                    currentNode = cache[indexRight]                                     # Let current node be the right child
                    k = 2*k + 2                                                         # Move node index to right child

                else:
                    # The root node reached the right position in the heap
                    break
```

```
            else:                                                                   # rightChild = None
                if int(currentNode.data) > int(leftChild.data):
                    # Swap cache objects id's to restore order
                    currentNode.label, cache[indexLeft].label = cache[indexLeft].label, currentNode.label
                    # Swap cache objects keys to restore order
                    currentNode.data, cache[indexLeft].data = cache[indexLeft].data, currentNode.data

                    # Correction of the chPos dictionary (label:data) pairs in parent node
                    if len(childKeys) == 2:
                        newPos = {currentNode.label : cache[indexLeft].label, childKeys[1] : childKeys[1]}
                        currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())
                    else:
                        newPos = {currentNode.label : cache[indexLeft].label}
                        currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())

                    # Correction of the chPos dictionary (label:data) pairs in previous node
                    if previousNode != None:
                        previousChildKeys = list(previousNode.chPos.keys())

                        if cache[indexLeft].label == previousChildKeys[0]:                  # If swapped node was a left child
                            newPos = {cache[indexLeft].label : currentNode.label, previousChildKeys[1] : previousChildKeys[1]}
                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())
                        else:                                                               # If swapped node was a right child
                            newPos = {previousChildKeys[0] : previousChildKeys[0], cache[indexLeft].label : currentNode.label}
                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())

                        previousNode = currentNode                                      # Assign current node to previousNode
                        currentNode = cache[indexLeft]                                  # Let current node be the left child
                        k = 2*k + 1                                                     # Move node index to left child

                break

            finalize('heap')

        return min

    else:
        return ()
```

After "Downheap" finishes, the cache contains all the nodes fetched by *readPath()* plus some nodes that were needed to restore the heap property. In this case, *finalize()* should accomplish the difficult task of attributing new random positions to all of them and of correctly updating all the chPos python dictionaries of the nodes in accordance with these new random values. This explains the different functionality of *finalize()* depending on its *typeIs* argument as it was pointed out in the previous chapter (note: in ***enqueue()***, the new random positions are created by the function itself).

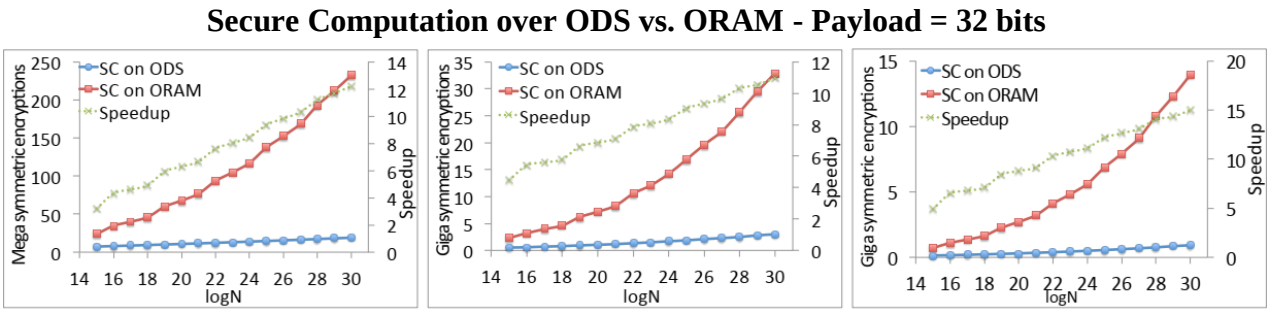
4.4 *Conclusions*

As stated by the authors of the original paper on which our work is based, the development of this general framework for oblivious data structures achieves better performance than generic ORAM both asymptotically and empirically (Fig. 11 and 12). This is a logical consequence of the fact that the predictability certain data structures exhibit in their access patterns can be efficiently used in designing their oblivious versions.



(a) Oblivious Map/Set (b) Oblivious Stack/Queue (c) Oblivious Deque/List

Fig. 11 [12]



(a) Oblivious Stack Operations (b) Oblivious Map/Set Operations (c) Oblivious Priority Queue Operations

Fig. 12 [12]

References

1. Alexander D'Agapeyeff (2013), *Codes and Ciphers - A History of Cryptography*.

2. Donald Davies, *A Brief History of Cryptography*. Information Security Technical Report. Vol. 2, No. 2 (1997) 14-17.

3. Huzaifa Sidhpurwala (2013), *A Brief History of Cryptography*, https://access.redhat.com/blogs/766093/posts/1976023

4. William A, Kotas (2000), *A Brief History of Cryptography*, University of Tennessee Honors Thesis Projects.

5. Gautham Sekar (2011), *Cryptanalysis and Design of Symmetric Cryptographic Algorithms*, Katholieke Universiteit Leuven – Faculty of Engineering.

6. Goldreich O. and Ostrovsky R. (1993), *Software Protection and Simulation on Oblivious RAMs*.

7. Elaine Shi, T-H. Hubert Chan, Emil Stefanov and Mingfei Li (2011), *Oblivious RAM with* $O((logN)^3)$ *Worst-Case Cost*.

8. Craig Gentry, Kenny Goldman, Shai Halevi and Charanjit Julta (2013), *Optimizing ORAM and Using it Efficiently for Secure Computation*.

9. Xiao Shaun Wang, T-H. Hubert Chan and Elaine Shi (2016), *Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound*.

10. Emil Stefanov, Marten van Dijkz, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu and Srinivas Devadas (2012), *Path ORAM: An Extremely Simple Oblivious RAM Protocol*.

11. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: *Oblivious ram for secure computation*. In ACM Conference on Computer and Communications Security (CCS), 2014.

12. Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang (2014), *Oblivious Data Structures*.

Appendix [Code listing]

odenode.py

```
class Odnode:
    def __init__(self, label, data, pos, chPos):
        self.label = label
        self.data = data
        self.pos = pos
        self.chPos = chPos
```

bintree.py

```
import sys
import math
import random
import crypt as cr
import json



class Node:
    def __init__(self, id, val):
        self.id = id
        self.value = val
        self.left = None
        self.right = None
        self.parent = None

    def set_left(self, nod):
        self.left = nod
        self.left.parent = self

    def set_right(self, nod):
        self.right = nod
        self.right.parent = self



class binTree(Node):
    def __init__(self, height, blocksPerBucket, passH):
        self.h = height
```

```
        self.z = blocksPerBucket
        self.key = passH
        self.nod = {}                                                   # Create a dictionary to hold the tree nodes
        self.emptyBucket = []                                           # Initialize empty bucket
        self.dummyLabel = cr.E(b'----------------\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10', self.key)
        self.dummyData = cr.E(b'----------------\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10', self.key)
        self.dummyPos = cr.E(b'9999999999999999\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10', self.key)
        self.dummyCPos = cr.E(b'----------------\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10', self.key)
        for k in range(self.z):                                         # Create an empty bucket of z dummy blocks
            self.emptyBucket.append((self.dummyLabel, self.dummyData, self.dummyPos, self.dummyCPos))


        for level in range(self.h + 1):                                 # Loop through the tree levels
            for i in range(2**level):                                   # Create 2^level nodes in every level
                self.nod[(level, i)] = Node((level, i), self.emptyBucket)   # Create a node holding an empty bucket at level 'level
'

                if level != 0:                                          # If the current node isn't the root do the following:
                    self.nod[(level, i)].parent = self.nod[(level-1, math.floor(i/2))]  # Set its parent node
                    if i%2 == 0:
                        self.nod[(level, i)].parent.set_left(self.nod[(level, i)])      #
                    else:                                               # Set its left and right children according to its position
                        self.nod[(level, i)].parent.set_right(self.nod[(level, i)])     #



    # Function P returns the path from a leaf-node to the root of the tree (a list of buckets)
    def P(self, leaf):
        self.currentNode = leaf
        self.path = []


        while self.currentNode.parent != None:                  # Until you reach the root DO:
            self.path.append(self.currentNode.value)            # Append current node's bucket to path (list)
            self.currentNode = self.currentNode.parent          # Move to parent node
        self.path.append(self.currentNode.value)                # Append root bucket to the list


        return self.path


    # Class method Pl returns the bucket id at level 'level' lying in the path from leaf-node 'leaf' to the root of the tree
    def Pl(self, leaf, level):
        self.currentNode = leaf
        self.currentLevel = self.h


        while self.currentLevel > level:                        # Until a certain level is reached DO:
            self.currentNode = self.currentNode.parent          # Move upwards in the leaf's path
            self.currentLevel -= 1


        bucket_id = self.currentNode.id                         # Get the id of the node at this level of leaf's path


        return bucket_id
```

crypt.py

```
from Crypto import Random as rnd
from Crypto.Hash import SHA256
from Crypto.Cipher import AES


def H(x):                                   # Method implementing SHA256 hash function
    key = bytes(str(x).encode('utf-8'))
    hash = SHA256.new()
    hash.digest_size = 16
    hash.update(key)
    return hash.digest()


def E(plaintext, key):                      # AES block cipher encryption method
    plain = plaintext
    k = key
    iv = rnd.new().read(AES.block_size)
    cipher = AES.new(k, AES.MODE_CBC, iv)
    return iv + cipher.encrypt(plain)


def D(ciphertext, key):
    ciphertext = ciphertext
    key = key
    iv = ciphertext[:AES.block_size]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext[AES.block_size:])
    return plaintext
```

ObliviousDataStructs.py

```
import os
import sys
import math
import random
import bintree as bt
import crypt as cr
import json
import odnode
```

```
password = 'myP@th0rAM'                          # Define the local password
passHash = cr.H(password)                        # Hash the local password in order to use it as a key for AES
Z = 4                                            # Define the number of blocks in a bucket



BS = 16                                                          #
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)   # pad and unpad methods used to match AES block size
unpad = lambda s: s[:-ord(s[len(s)-1:])]                         #



def oramAccess(op, block_node):
    global S
    S = []                                       # Initialize local stash as a list of tuples
    oramPath = []

    if op != 'readandremove' and op != 'add': raise ValueError

    def writeBucket(bucketID, block_list):
        while len(block_list) < Z:                                                                  # Pad the bucket with dummy ..
            block_list.append(('----------------', '----------------', '9999999999999999', '----------------')) # .. blocks until its size is Z

        # Encrypt the bucket blocks to be written in the ORAM
        enBucket = [(cr.E(bytes(pad(bl[0]).encode('utf-8')), passHash),
        cr.E(bytes(pad(bl[1]).encode('utf-8')), passHash),
        cr.E(bytes(pad(str(bl[2])).encode('utf-8')), passHash),
        cr.E(bytes(pad(str(bl[3])).encode('utf-8')), passHash)) for bl in block_list]
        # Write the encrypted bucket in the ORAM
        oram.nod[bucketID].value = enBucket

    jnode = json.dumps(block_node.__dict__)          # Serialize object block_node to JSON
    dnode = json.loads(jnode)                        # Turn JSON into python dictionary

    x = dnode['pos']
    oramPath = oram.P(oram.nod[L, x])                # Get the path of leaf x and store it locally in a list of buckets

    # Add to the local stash S the decrypted blocks of the oramPath list
    for l in range(L+1):
        for b in range(Z):
            blockContent = (unpad(cr.D(oramPath[l][b][0], passHash).decode('utf-8')),
            unpad(cr.D(oramPath[l][b][1], passHash).decode('utf-8')),
            int(unpad(cr.D(oramPath[l][b][2], passHash).decode('utf-8'))),
            unpad(cr.D(oramPath[l][b][3], passHash).decode('utf-8')))
            if blockContent[0] != '----------------':
                S.append(blockContent)

    block = next((a for a in S if a[0] == dnode['label']), ('None', 'Null', 0, {}))    # Read the block in question from the local stash

    if op == 'add':                                                                    # If the operation is 'add':
        print('(Add)')
```

```
        if block in S:
            S.remove(block)                                                 # Remove the old block from the stash if it's there
        S.append((dnode['label'], dnode['data'], dnode['pos'], dnode['chPos']))     # Add the new block, data and its children positions or the
 old block with new data


    if block in S:
        S.remove(block)
    S_temp = []
    for l in range(L, -1, -1):
        # S_temp = {b in S : P(x, l) = P(position[b], l)}
        S_temp = [b for b in S if oram.Pl(oram.nod[L, x], l) == oram.Pl(oram.nod[L, b[2]], l)]
        # S_temp = {Select min(|S_temp|, Z) elements from S_temp}
        S_temp = S_temp[:min(len(S_temp), Z)]
        S = [item for item in S if item not in S_temp]                                  # S = S - S_temp
        writeBucket(oram.Pl(oram.nod[L, x], l), S_temp)                                 # WriteBucket(P(x, l), S_temp)


    if op == 'readandremove':
        print('(ReadAndRemove)')
        askedBlock = odnode.Odnode(block[0], block[1], block[2], json.loads(str(block[3]).replace("'", '"')))
        return askedBlock




##########################  Initial data entry functions  ##########################


def dataInputStack(N):
    print('\n\nInitial data entry')
    print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')


    global root
    global top


    for i in range(N):
        blockLabel = input('Label of {} No. {}: '.format(blockAlias, i))
        blockData = input('Data of {} No. {}: '.format(blockAlias, i))
        print()
        pos = random.randint(0, 2**L - 1)                           # Generate random positions for the nodes
        blkNode = odnode.Odnode(blockLabel, blockData, pos, {})     # Create instance of Odnode class and assign the values of the current bloc
k (node)
        blocks.append(blkNode)                                      # Construct a list holding the data blocks (nodes)




    # Store children's positions in a dictionary for each node
    for i, j in enumerate(blocks):
        if i < len(blocks)-1:
            cName = blocks[i+1].label                               # Assign to cName current block's child label
            cPos = blocks[i+1].pos                                  # Assign to cPos current block's child position
            j.chPos = {cName : cPos}                                # Add to current block the pair {Child_id : position}
```

```
            if i == 0:                                          # Store the root of the ..
                root = j                                        # .. data structure in variable 'root'
        top = blocks[-1]                                        # Store last node of the list in 'top'

        # Write given data blocks in ORAM
        for k in blocks:
            oramAccess('add', k)



#################################################################

def dataInputQueue(N):
    print('\n\nInitial data entry')
    print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')

    global root
    global top
    global nextID
    global nextPOS
    global queueSize

    queueSize = N

    for i in range(N):
        blockData = input('Data of {} No. {}: '.format(blockAlias, i))
        print()
        pos = random.randint(0, 2**L - 1)                   # Generate random positions for the nodes
        blkNode = odnode.Odnode(str(i), blockData, pos, {}) # Create instance of Odnode class and assign the values of the current block (node)
        blocks.append(blkNode)                              # Construct a list holding the data blocks (nodes)
    nextID = str(N)
    nextPOS = random.randint(0, 2**L - 1)                   # Generate an extra random position to be used in enqueue()

    # Store children's positions in a dictionary for each node
    for i, j in enumerate(blocks):
        if i < len(blocks)-1:
            cName = blocks[i+1].label                               # Assign to cName current block's child label
            cPos = blocks[i+1].pos                                  # Assign to cPos current block's child position
            j.chPos = {cName : cPos}                                # Add to current block the pair {Child_id : position}
            if i == 0:                                              # Store the root of the ..
                root = j                                            # .. data structure in variable 'root'
    top = blocks[-1]                                                # Store last node of the list in 'top'
    top.chPos = {nextID : nextPOS}

    # Write given data blocks in ORAM
    for k in blocks:
        oramAccess('add', k)
```

```
################################################################


def heapify(nodelist, index, N):
    left = 2*index + 1
    right = 2*index + 2

    if (left < N) and (int(nodelist[left].data) < int(nodelist[index].data)):
        smallest = left
    else:
        smallest = index

    if (right < N) and (int(nodelist[right].data) < int(nodelist[smallest].data)):
        smallest = right

    if smallest != index:
        nodelist[index], nodelist[smallest] = nodelist[smallest], nodelist[index]   # Perform swap if needed
        heapify(nodelist, smallest, N)
    blocks[:] = nodelist


################################################################


def dataInputHeap(N):
    print('\n\nInitial data entry')
    print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')

    global root
    global last
    for i in range(N):
        blockLabel = input('ID of {} No. {}: '.format(blockAlias, i))
        blockData = input('Key of {} No. {}: '.format(blockAlias, i))
        print()
        pos = random.randint(0, 2**L - 1)
        blkNode = odnode.Odnode(blockLabel, blockData, pos, {})     # Create instance of Odnode class and assign the values of the current bloc
k (node)
        blocks.append(blkNode)                                      # Construct a list holding the data blocks (nodes)



    # Calculate the last parent node depending on input size
    if len(blocks) > 1:
        lastParent = math.floor(len(blocks)/2) - 1
    else:
        lastParent = -1
        root = blocks[0]

    # Heapify
    for i in range(lastParent, -1, -1):
        heapify(blocks, i, N)
```

```
    last = len(blocks)


    # Store children's positions in a dictionary for each node
    for i, j in enumerate(blocks):
        if i <= lastParent:                                     # Until we reach the index of the parent of last node in the heap

            indexLeft = 2*i + 1                                 # Calculate the index of left child
            indexRight = 2*i + 2                                # Calculate the index of right child
            cLName = blocks[indexLeft].label                    # Assign to cLName current block's Left child label
            cLPos = blocks[indexLeft].pos                       # Assign to cLPos current block's Left child position
            if indexRight < len(blocks):
                cRName = blocks[indexRight].label               # Assign to cRName current block's Right child label
                cRPos = blocks[indexRight].pos                  # Assign to cRPos current block's Right child position
                j.chPos = {cLName : cLPos, cRName : cRPos}      # Add to current block the children positions dictionary
            else:
                j.chPos = {cLName : cLPos}                      # The current node has only a left child
            if i == 0:                                          # Store the root of the ..
                root = j                                        # .. data structure in variable 'root'

    # Write given data blocks in ORAM
    for k in blocks:
        oramAccess('add', k)


############################  Path ORAM Explorer  ############################
def oramExplorer():
    while True:
        os.system('clear')
        print('\n\nPath ORAM explorer')
        print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u
203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\
u203e\u203e\u203e\u203e\u203e\u203e')
        print("[1] --> Display path ORAM's content (Decrypted)")
        print("\n[2] --> Display path ORAM's raw content (Encrypted)")
        print('---------------------------------------------------')
        print('[ENTER] --> EXIT')
        print('___________________________________________________\n\n\n')

        com = input('Please enter your choice : ')

        if com == '':
            return

        ###################  Display the ORAM (Decrypted)  ###################
        elif com == '1':
            print()
            for k in sorted(oram.nod.keys()):
                print('\nBucket id =', k)
```

```
                blst = [(unpad(cr.D(oram.nod[k].value[i][0], passHash).decode('utf-8')),
                unpad(cr.D(oram.nod[k].value[i][1], passHash).decode('utf-8')),
                int(unpad(cr.D(oram.nod[k].value[i][2], passHash).decode('utf-8'))),
                unpad(cr.D(oram.nod[k].value[i][3], passHash).decode('utf-8'))) for i in range(Z)]
                print(blst)
            input('\nPlease press [ENTER] to continue...')



        ###################  Display the ORAM (Encrypted)  ###################
        elif com == '2':
            print()
            for k in sorted(oram.nod.keys()):
                print('\nBucket id =', k)
                blst = [(oram.nod[k].value[i][0].hex(), oram.nod[k].value[i][1].hex(),
                (oram.nod[k].value[i][2].hex(), oram.nod[k].value[i][3].hex())) for i in range(Z)]
                print(blst)
            input('\nPlease press [ENTER] to continue...')




############################   ODS Framework Functions   ############################


def odsStart():                             # Update cache to contain the root
    global cache
    global root
    cache.clear()
    if root != None:
        cache.append(root)



###################  Read  ###################

def read(nodeLabel):
    global cache

    isInCache = any(x.label == nodeLabel for x in cache)                        # True if the block in question is already in cache
    if isInCache == False:
        n = 0
        while isInCache == False:                                               #
            childDictKeys = list(cache[n].chPos.keys())                         #
            childName = childDictKeys[0]                                        #
            childPosition = cache[n].chPos[childName]                           # Traverse through the nodes ..
            if not any(x.label == childName for x in cache):                    # .. using their children positions ..
                ask = odnode.Odnode(childName, 'null', childPosition, {})       # .. until the requested one is found.
                fetch = oramAccess('readandremove', ask)                        #
                cache.append(fetch)                                             #
            isInCache = any(x.label == nodeLabel for x in cache)                #
            n += 1                                                              #
    return cache[-1]                                                            # Return the last object (node) in cache
```

```
###################  Insert  ###################

def insert(newNodeLabel, newNodeData):
    global cache
    global root
    global top

    newNode = odnode.Odnode(newNodeLabel, newNodeData, 0, {})       # Create Odnode instance for the new block (node)
    if root != None: oramAccess('readandremove', root)              # Remove root from ORAM
    cache.insert(0, newNode)                                        # Insert new block in cache at index 0
    root = newNode                                                  # newNode is the new root
    if top == None:                                                 # If the structure was empty, newNode is also the new top
        top = newNode



################   Update(Write)   ################

def write(nodeLabel, newData):
    global cache
    isInCache = any(x.label == nodeLabel for x in cache)               # True if the block in question is already in cache
    if isInCache == False:
        read(nodeLabel)
    next((n for n in cache if n.label == nodeLabel)).data = newData



#################   Delete   #################

def delete(nodeLabel):
    global cache
    global root

    if len(cache) == 0:
        print('\nThe Oblivious Data Structure is empty!\n')

    else:
        if root != None: oramAccess('readandremove', root)                 # Remove root from ORAM
        read(nodeLabel)                                                    # Get the node from ORAM
        del cache[-1]                                                      # Delete from cache



################   Finalize   ################

def finalize(typeIs):
    print('finalize()')
    global cache
    global root
    global top
```

```
if cache != [] and typeIs != 'enqueue':
    # Assign new random position to each node in cache
    for n in cache:
        pos = random.randint(0, 2**L - 1)
        n.pos = pos

    # Update children's positions in each node
    if typeIs == 'linear':
        for i, j in enumerate(cache):
            if i < len(cache)-1:
                cName = cache[i+1].label                    # Assign to cName current block's child label
                cPos = cache[i+1].pos                       # Assign to cPos current block's child position
                j.chPos = {cName : cPos}                    # Add to current block the pair {Child_id : position}
                if i == 0:                                  # Store the root of the ..
                    root = j                                # .. data structure in variable 'root'



    if typeIs == 'heap':
        cacheNodeDict = dict((x.label, x.pos) for x in cache)

        for i, j in enumerate(cache):
            if i < len(cache)-1:
                childrenList = list(j.chPos.keys())

                if len(childrenList) > 0:
                    if childrenList[0] in list(cacheNodeDict.keys()):
                        j.chPos[childrenList[0]] = cacheNodeDict[childrenList[0]]

                if len(childrenList) > 1:
                    if childrenList[1] in list(cacheNodeDict.keys()):
                        j.chPos[childrenList[1]] = cacheNodeDict[childrenList[1]]

                if i == 0:                                          # Store the root of the ..
                    root = j                                        # .. data structure in variable 'root'

# Write cahe back to ORAM
for k in cache:
    oramAccess('add', k)

# Empty client cache
cache.clear()
```

```
###########################################  Main program loop  ###########################################
while True:

    blocks = []                                     # Initialize the list holding the data blocks (nodes)
    cache = []                                      # Initialize local (client) cache

    os.system('clear')
    print('\n\nODS (Oblivious Data Structure) CREATION')
    print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e
\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')
    print('[1] --> Oblivious Stack')
    print('\n[2] --> Oblivious Queue')
    print('\n[3] --> Oblivious Heap')
    print('---------------------------------------')
    print('[ENTER] --> EXIT')
    print('_______________________________________\n\n\n')

    oblStruct = input('Please enter your choice : ')

    if oblStruct == '1' or oblStruct == '2' or oblStruct == '3':
        N = int(input('\nInitial number of items/nodes (>1) : '))       # Get the number of items the stack will contain
        L = math.ceil(math.log(N, 2))                               # Calculate path-oram's tree height L
        oram = bt.binTree(L, Z, passHash)                           # Construct an instance of the binTree class with the given parametres

    if oblStruct == '1':
        cache.clear()
        os.system('clear')
        blockAlias = 'item'
        print('\n\nOBLIVIOUS STACK')
        print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')
        dataInputStack(N)

        while True:
            # present stack menu
            os.system('clear')

            print('\n\nOblivious Stack Options')
            print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u20
3e\u203e\u203e\u203e\u203e\u203e\u203e')
            print('[1] --> Push(item)')
            print('\n[2] --> Pop()')
            print('\n[3] --> IsEmpty()')
            print('---------------------------')
            print('[4] --> Path ORAM explorer')
            print('\n[ENTER] --> EXIT')
            print('___________________________\n\n\n')

            select = input('Please enter your choice : ')
```

```
if select == '':
    break


odsStart()
if select == '1':
    newBlockName = input('\nEnter the ID of the item you want to push : ')
    newBlockData = input("Enter the data of item '{0}' : ".format(newBlockName))
    print()


    def push(node, data):
        insert(node, data)
        finalize('linear')


    push(newBlockName, newBlockData)


    print('\nOperation finished successfully!')
    input('\nPlease press [ENTER] to continue...')



if select == '2':
    print()


    def pop():
        global cache
        global root


        if root != None:
            oldTop = oramAccess('readandremove', root)


            if oldTop.chPos == {}:                                  # If this is the last item
                newRoot = None
            else:
                rootChildKey = list(oldTop.chPos.keys())[0]         # Get the root's child label
                newRoot = read(rootChildKey)                        # Read root's next item
                del cache[0]                                        # Delete old root (top)
            root = newRoot
            finalize('linear')
        else:
            oldTop = None


        return oldTop


    topItem = pop()


    if topItem != None:
        print('\nItem ID :', topItem.label)
        print('Item Data :', topItem.data)
```

```
                else:
                    print('\nThe Oblivious Stack is empty!')

                input('\nPlease press [ENTER] to continue...')


            if select == '3':

                def isStackEmpty():
                    return (len(cache) == 0)

                ans = isStackEmpty()
                if ans:
                    print('\nTRUE - The Oblivious Stack is empty.')
                else:
                    print('\nFALSE - The Oblivious Stack is NOT empty.')

                input('\nPlease press [ENTER] to continue...')


            if select == '4':
                oramExplorer()



    if oblStruct == '2':
        cache.clear()
        os.system('clear')
        blockAlias = 'item'
        print('\n\nOBLIVIOUS QUEUE')
        print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')
        dataInputQueue(N)

        while True:
            # present queue menu
            os.system('clear')

            print('\n\nOblivious Queue Options')
            print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')
            print('[1] --> Enqueue(item)')
            print('\n[2] --> Dequeue()')
            print('\n[3] --> IsEmpty()')
            print('---------------------------')
            print('[4] --> Path ORAM explorer')
            print('\n[ENTER] --> EXIT')
            print('___________________________\n\n\n')
```

```
        select = input('Please enter your choice : ')


        if select == '':
            break


        odsStart()
        if select == '1':
            global nextID
            global nextPOS
            global queueSize


            newID = nextID
            newPOS = nextPOS


            queueSize += 1
            nextID = str(int(newID) + 1)
            nextPOS = random.randint(0, 2**L - 1)               # Generate an extra random position for next enqueue()


            newBlockData = input("\nEnter the data of item '{0}' : ".format(newID))
            print()
            newNode = odnode.Odnode(newID, newBlockData, newPOS, {nextID : nextPOS})


            def enqueue(qnode):
                global root
                global queueSize


                cache.clear()
                cache.append(qnode)
                finalize('enqueue')
                if queueSize == 1:
                    root = newNode


            enqueue(newNode)


            print('\nOperation finished successfully!')
            input('\nPlease press [ENTER] to continue...')




        if select == '2':
            print()


            def dequeue():
                global cache
                global root
                global queueSize


                if root != None:
                    oldHead = oramAccess('readandremove', root)
```

```
                    queueSize -= 1
                    rootChildKey = list(oldHead.chPos.keys())[0]

                    if queueSize == 0:                          # If this is the last item
                        newRoot = None
                    else:
                        newRoot = read(rootChildKey)            # Read root's next item
                        del cache[0]                            # Delete old root (top)
                    root = newRoot
                    finalize('linear')
                else:
                    oldHead = None

                return oldHead

            headItem = dequeue()

            if headItem != None:
                print('\nItem ID :', headItem.label )
                print('Item Data :', headItem.data)

            else:
                print('\nThe Oblivious Queue is empty!')

            input('\nPlease press [ENTER] to continue...')


        if select == '3':

            def isQueueEmpty():
                return (len(cache) == 0)

            ans = isQueueEmpty()

            if ans:
                print('\nTRUE - The Oblivious Queue is empty.')
            else:
                print('\nFALSE - The Oblivious Queue is NOT empty.')

            input('\nPlease press [ENTER] to continue...')


        if select == '4':
            oramExplorer()


if oblStruct == '3':
    cache.clear()
```

```
        os.system('clear')
        blockAlias = 'element'
        print('\n\nOBLIVIOUS HEAP (PRIORITY QUEUE)')
        print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u
203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e')
        dataInputHeap(N)


        def readPath(operation):
            global cache
            global root
            global last

            depth = math.floor(math.log2(last))
            currentNode = oramAccess('readandremove', root)

            binLast = last + 1

            if ((binLast & (binLast - 1)) != 0) or (operation == 'extract'):                # Last node IS NOT at the end of a tree row ..
                for k in range(depth-1, 0, -1):                                           # .. or the call is from extractMin()
                    # Check if the last parent has already 2 children. If yes, go to the next
                    if k == 1 and (last % 2 == 1) and (operation == 'insert'):
                        ind = math.floor(last/2)
                    else:
                        ind = math.floor(last/math.pow(2,k)) - 1

                    if (ind % 2) == 1:
                        leftChildLabel = list(currentNode.chPos.keys())[0]
                        leftChildPos = currentNode.chPos[leftChildLabel]
                        ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})       # Ask for the left child ..
                        fetch = oramAccess('readandremove', ask)                           # .. to be fetched from ORAM
                    else:
                        rightChildLabel = list(currentNode.chPos.keys())[1]
                        rightChildPos = currentNode.chPos[rightChildLabel]
                        ask = odnode.Odnode(rightChildLabel, 'null', rightChildPos, {})     # Ask for the right child ..
                        fetch = oramAccess('readandremove', ask)                           # .. to be fetched from ORAM

                    cache.append(fetch)                                                    # Append fetched node to cache
                    currentNode = fetch
            else:                                                                          # Last node IS at the end of a tree row
                for k in range(depth):
                    leftChildLabel = list(currentNode.chPos.keys())[0]
                    leftChildPos = currentNode.chPos[leftChildLabel]
                    ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})           # Ask for the left child ..
                    fetch = oramAccess('readandremove', ask)                               # .. to be fetched from ORAM
                    cache.append(fetch)                                                    # Append fetched node to cache
                    currentNode = fetch
```

```
        while True:
            # present heap menu
            os.system('clear')

            print('\n\nOblivious Heap Options')
            print('\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u203e\u20
3e\u203e\u203e\u203e\u203e\u203e\u203e')
            print('[1] --> Insert(element)')
            print('\n[2] --> ExtractMin()')
            print('\n[3] --> IsEmpty()')
            print('--------------------------')
            print('[4] --> Path ORAM explorer')
            print('\n[ENTER] --> EXIT')
            print('___________________________\n\n\n')

            select = input('Please enter your choice : ')

            if select == '':
                break

            odsStart()
            if select == '1':
                newBlockName = input('\nEnter the ID of the element you want to insert : ')
                newBlockData = input("Enter the key of element '{0}' : ".format(newBlockName))
                print()

                def insertKey(id, key):
                    global cache
                    global root
                    global last

                    newNode = odnode.Odnode(id, key, 0, {})                                      # Create Odnode instance for the new node

                    if last == 0:
                        root = newNode
                        oramAccess('add', newNode)
                        last = 1
                    else:
                        readPath('insert')
                        cache.append(newNode)                                                    # Append new node in cache
                        cache[-2].chPos[cache[-1].label] = cache[-1].pos                         # Attach new node to the heap

                        #################################  Upheap  #################################
                        k = len(cache)-1
                        while (k > 0) and (int(cache[k].data) < int(cache[k-1].data)):
                            cache[k-1].label, cache[k].label = cache[k].label, cache[k-1].label   # Swap cache objects id's to restore order
                            cache[k-1].data, cache[k].data = cache[k].data, cache[k-1].data       # Swap cache objects keys to restore order
```

```
            childKeys = list(cache[k-1].chPos.keys())

            ######################################################################## If swapped node was left child
            if childKeys[0] == cache[k-1].label:
                if len(childKeys) == 2:
                    newPos = {cache[k-1].label : cache[k].label, childKeys[1] : childKeys[1]}
                    cache[k-1].chPos = dict((newPos[key], value) for (key, value) in cache[k-1].chPos.items())
                else:
                    newPos = {cache[k-1].label : cache[k].label}
                    cache[k-1].chPos = dict((newPos[key], value) for (key, value) in cache[k-1].chPos.items())

                # If swapped node hasn't reached the root
                if k-2 >= 0:
                    childKeysParent = list(cache[k-2].chPos.keys())

                    # If swapped node was left child
                    if childKeysParent[0] == cache[k].label:
                        newPosParent = {cache[k].label : cache[k-1].label, childKeysParent[1] : childKeysParent[1]}
                        cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

                    # If swapped node was right child
                    if childKeysParent[1] == cache[k].label:
                        newPosParent = {childKeysParent[0] : childKeysParent[0], cache[k].label : cache[k-1].label}
                        cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

            ######################################################################## If swapped node was right child
            if len(childKeys) > 1 and childKeys[1] == cache[k-1].label:
                newPos = {childKeys[0] : childKeys[0], cache[k-1].label : cache[k].label}
                cache[k-1].chPos = dict((newPos[key], value) for (key, value) in cache[k-1].chPos.items())

                # If swapped node hasn't reached the root
                if k-2 >= 0:
                    childKeysParent = list(cache[k-2].chPos.keys())

                    # If swapped node was left child
                    if childKeysParent[0] == cache[k].label:
                        newPosParent = {cache[k].label : cache[k-1].label, childKeysParent[1] : childKeysParent[1]}
                        cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

                    # If swapped node was right child
                    if childKeysParent[1] == cache[k].label:
                        newPosParent = {childKeysParent[0] : childKeysParent[0], cache[k].label : cache[k-1].label}
                        cache[k-2].chPos = dict((newPosParent[key], value) for (key, value) in cache[k-2].chPos.items())

            k -= 1

        last += 1
        finalize('heap')
```

```
        insertKey(newBlockName, newBlockData)

        print('\nOperation finished successfully!')
        input('\nPlease press [ENTER] to continue...')


    if select == '2':
        print()

        def extractMin():
            global cache
            global root
            global last

            if last > 0:
                readPath('extract')
                currentNode = cache[-1]

                # If last node in cache is not a leaf, fetch another one
                if currentNode.chPos != {}:
                    ind = last - 1

                    if (ind % 2) == 1:
                        leftChildLabel = list(currentNode.chPos.keys())[0]
                        leftChildPos = currentNode.chPos[leftChildLabel]
                        ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})       # Ask for the left child ..
                        fetch = oramAccess('readandremove', ask)                            # .. to be fetched from ORAM
                    else:
                        rightChildLabel = list(currentNode.chPos.keys())[1]
                        rightChildPos = currentNode.chPos[rightChildLabel]
                        ask = odnode.Odnode(rightChildLabel, 'null', rightChildPos, {})     # Ask for the right child ..
                        fetch = oramAccess('readandremove', ask)                            # .. to be fetched from ORAM

                    cache.append(fetch)                                                     # Append fetched node to cache

                min = (cache[0].label, cache[0].data)                                       # Assign minimum element to min
                cache[0].label = cache[-1].label                                            # Last element becomes the new ..
                cache[0].data  = cache[-1].data                                             # .. root leaving chPos's as they are
                del cache[-1]                                                               # Remove last element from cache

                if len(cache) > 0:
                    del cache[-1].chPos[cache[0].label]           # Remove previous last element from its parent's chPos dictionary

                last -= 1


                ################################  Downheap  ################################
```

```
if last > 0:
    currentNode = cache[0]
    previousNode = None

    k = 0
    while k <= math.floor(last/2) - 1:
        childKeys = list(currentNode.chPos.keys())

        if len(childKeys) > 0:                                          # If current node has at least 1 child
            leftChildLabel = childKeys[0]
            leftChildPos = currentNode.chPos[leftChildLabel]

            isInCache = any(x.label == leftChildLabel for x in cache)          # True if this node is already in cache
            if isInCache == False:
                ask = odnode.Odnode(leftChildLabel, 'null', leftChildPos, {})           # Ask for the left child ..
                leftChild = oramAccess('readandremove', ask)                            # .. to be fetched from ORAM
                cache.append(leftChild)                                                 # Add left child to cache
            else:
                leftChild = next(n for n in cache if n.label == leftChildLabel)
            indexLeft = next((i for i, item in enumerate(cache) if item.label == leftChildLabel), -1)

            rightChild = None                                                   # Initialize right child node



        if len(childKeys) > 1:                                              # If current node has 2 children
            rightChildLabel = childKeys[1]
            rightChildPos = currentNode.chPos[rightChildLabel]

            isInCache = any(x.label == rightChildLabel for x in cache)         # True if this node is already in cache
            if isInCache == False:
                ask = odnode.Odnode(rightChildLabel, 'null', rightChildPos, {})         # Ask for the right child ..
                rightChild = oramAccess('readandremove', ask)                           # .. to be fetched from ORAM
                cache.append(rightChild)                                                # Add right child to cache
            else:
                rightChild = next(n for n in cache if n.label == rightChildLabel)
            indexRight = next((i for i, item in enumerate(cache) if item.label == rightChildLabel), -1)

        if len(childKeys) == 0:
            # The node has no children
            break



        if rightChild != None:
            if int(leftChild.data) < int(rightChild.data):
                if int(currentNode.data) > int(leftChild.data):
                    # Swap cache objects id's to restore order
                    currentNode.label, cache[indexLeft].label = cache[indexLeft].label, currentNode.label
                    # Swap cache objects keys to restore order
                    currentNode.data, cache[indexLeft].data = cache[indexLeft].data, currentNode.data
```

```
                                    # Correction of the chPos dictionary (label:data) pairs in current node

                                    if len(childKeys) == 2:
                                        newPos = {currentNode.label : cache[indexLeft].label, childKeys[1] : childKeys[1]}
                                        currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())
                                    else:
                                        newPos = {currentNode.label : cache[indexLeft].label}
                                        currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())

                                    # Correction of the chPos dictionary (label:data) pairs in previous node
                                    if previousNode != None:
                                        previousChildKeys = list(previousNode.chPos.keys())

                                        if cache[indexLeft].label == previousChildKeys[0]:         # If swapped node was a left child
                                            newPos = {cache[indexLeft].label : currentNode.label, previousChildKeys[1] : previousChildK
eys[1]}

                                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items
())
                                        else:                                                      # If swapped node was a right child
                                            newPos = {previousChildKeys[0] : previousChildKeys[0], cache[indexLeft].label : currentNode
.label}

                                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items
())

                                    previousNode = currentNode                                  # Assign current node to previousNode
                                    currentNode = cache[indexLeft]                              # Let current node be the left child
                                    k = 2*k + 1                                                 # Move node index to left child

                                else:
                                    # The root node reached the right position in the heap
                                    break


                            else:
                                if int(currentNode.data) > int(rightChild.data):
                                    # Swap cache objects id's to restore order
                                    currentNode.label, cache[indexRight].label = cache[indexRight].label, currentNode.label
                                    # Swap cache objects keys to restore order
                                    currentNode.data, cache[indexRight].data = cache[indexRight].data, currentNode.data

                                    # Correction of the chPos dictionary (label:data) pairs in current node

                                    newPos = {childKeys[0] : childKeys[0], currentNode.label : cache[indexRight].label}
                                    currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())

                                    # Correction of the chPos dictionary (label:data) pairs in previous node
                                    if previousNode != None:
```

```
                                        previousChildKeys = list(previousNode.chPos.keys())


                                        if cache[indexRight].label == previousChildKeys[0]:         # If swapped node was a left child
                                            newPos = {cache[indexRight].label : currentNode.label, previousChildKeys[1] : previousChild
Keys[1]}
                                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items
())
                                        else:                                                       # If swapped node was a right child
                                            newPos = {previousChildKeys[0] : previousChildKeys[0], cache[indexRight].label : currentNod
e.label}
                                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items
())


                                    previousNode = currentNode                                  # Assign current node to previousNode
                                    currentNode = cache[indexRight]                             # Let current node be the right child
                                    k = 2*k + 2                                                 # Move node index to right child


                                else:
                                    # The root node reached the right position in the heap
                                    break



                            else:                                                               # rightChild = None
                                if int(currentNode.data) > int(leftChild.data):
                                    # Swap cache objects id's to restore order
                                    currentNode.label, cache[indexLeft].label = cache[indexLeft].label, currentNode.label
                                    # Swap cache objects keys to restore order
                                    currentNode.data, cache[indexLeft].data = cache[indexLeft].data, currentNode.data


                                    # Correction of the chPos dictionary (label:data) pairs in parent node


                                    if len(childKeys) == 2:
                                        newPos = {currentNode.label : cache[indexLeft].label, childKeys[1] : childKeys[1]}
                                        currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())
                                    else:
                                        newPos = {currentNode.label : cache[indexLeft].label}
                                        currentNode.chPos = dict((newPos[key], value) for (key, value) in currentNode.chPos.items())


                                    # Correction of the chPos dictionary (label:data) pairs in previous node
                                    if previousNode != None:
                                        previousChildKeys = list(previousNode.chPos.keys())


                                        if cache[indexLeft].label == previousChildKeys[0]:          # If swapped node was a left child
                                            newPos = {cache[indexLeft].label : currentNode.label, previousChildKeys[1] : previousChildKeys[
1]}
                                            previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())
                                        else:                                                       # If swapped node was a right child
```

```
                                        newPos = {previousChildKeys[0] : previousChildKeys[0], cache[indexLeft].label : currentNode.lab
el}
                                        previousNode.chPos = dict((newPos[key], value) for (key, value) in previousNode.chPos.items())

                                    previousNode = currentNode                              # Assign current node to previousNode
                                    currentNode = cache[indexLeft]                          # Let current node be the left child
                                    k = 2*k + 1                                             # Move node index to left child
                                break
                            finalize('heap')
                        return min
                    else:
                        return ()

                heapMin = extractMin()

                if heapMin != ():
                    print('\nMinimun =', heapMin)
                else:
                    print('\nThe Oblivious Queue is empty!')

                input('\nPlease press [ENTER] to continue...')



            if select == '3':

                def isEmpty():
                    return (last == 0)

                ans = isEmpty()

                if ans:
                    print('\nTRUE - The Oblivious Heap is empty.')
                else:
                    print('\nFALSE - The Oblivious Heap is NOT empty.')

                input('\nPlease press [ENTER] to continue...')



            if select == '4':
                oramExplorer()



    if oblStruct == '':
        break
```