# AN ABSTRACT OF THE THESIS OF

Gabriel A. Hackebeil for the degree of Master of Science in Computer Science presented on June 3, 2016.

Title: Efficient Oblivious Access to Trees

Abstract approved: _____

Attila A. Yavuz

The outsourcing of data storage and related infrastructure to third-party services in the cloud is a trend that has gained considerable momentum in the last decade due to the savings it affords companies in both capital and operational costs. Although encryption can alleviate some of the privacy concerns associated with cloud storage, it comes at the cost of decreased utility of data once it is in the cloud. For instance, cloud services for searching over a large set of files are useless when those files are encrypted using standard, randomized techniques. Moreover, even though the files are encrypted, the historical access pattern over a data set (e.g., file access times/frequency, memory read/write locations) can leak significant information to malicious parties that can be combined with other *metadata* to partially reveal the file contents.

Two research thrusts that address these issues are *Symmetric Searchable Encryption* (SSE) and *Oblivious RAM* (ORAM). SSE schemes provide an efficient means for searching over encrypted data that is stored on an untrusted server. These schemes, however, are inherently susceptible to statistical attacks by observing the history of deterministically encrypted search queries, along with the access pattern resulting from

those queries. ORAM, by design, can prevents such leakages, but it introduces high communication overhead, among other challenges.

In this work we introduce new ORAM schemes tailored for more efficient private access to tree data structures, which are commonly used as search indexes. We make multi-faceted contributions that include: (i) a formal definition for obliviousness in the relaxed setting where the external storage provider knows we are accessing a tree data structure, (ii) a formal proof showing that our schemes satisfy this definition, with analytical and empirical results showing that we reduce transmission overhead by several factors over state-of-the-art ORAM schemes, and (iii) a Python software package called PyORAM (Python-based Oblivious RAM) that provides researchers with a powerful set of tools to build upon current ORAM methods tailored for the cloud storage setting. Our work has broad impacts for searchable encryption via private database search with reduced transmission requirements, which is vital for privacy-critical applications in governmental and healthcare systems.

# Efficient Oblivious Access to Trees

by

Gabriel A. Hackebeil

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 3, 2016
Commencement June 2016

Master of Science thesis of Gabriel A. Hackebeil presented on June 3, 2016.

APPROVED:

---

Major Professor, representing Computer Science

---

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

---

Gabriel A. Hackebeil, Author

# ACKNOWLEDGEMENTS

First and foremost, I would like to thank my partner in life and best friend Laura Hirshfield for her endless patience, love, and support during my time as a student.

I would also like to express my sincere gratitude to my academic advisor Professor Attila Yavuz, Associate Professor in EECS at OSU, for giving me the guidance and support to carry out this research.

Finally, I would like to thank my committee members Professor Rakesh Bobba, Professor Ken Funk, and Professor Mike Rosulek for accepting to serve on my committee and for the time they devoted to reading this thesis.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# Chapter 1: Introduction

## 1.1 Problem Statement

Cloud computing and storage has become a critical service for many companies over the last decade as it offers considerable savings in both capital and operational costs. Companies that utilize cloud services have lower startup costs because there is no need to house server equipment, reduced energy consumption because service providers can more efficiently manage bulk server energy requirements, and a flexible resource model that can quickly scale with changing demands. At the forefront of these online services are technology giants such as Amazon, Microsoft, Oracle, IBM, and Google.

The use of cloud services also introduces a number of privacy concerns. Using cloud services for data storage and backup increases the attack surface area over which malicious parties can attempt to gain access to a client's data. Data that is placed in the cloud is necessarily distributed and replicated on multiple physical storage media. This may be the result of data resiliency guarantees offered by the service provider, or simply due to the distributed nature of this system. Either way, there is an increased risk of unauthorized physical access to the data. Additionally, the number of individuals that can potentially be bribed or coerced into providing unauthorized access to the data greatly increases as it now includes a potentially large set of employees or contractors associated with the service provider. There are also legal questions regarding ownership of the data once it is stored in the cloud. What rights does a service provider have when it comes to sharing the data with third-parties? A definite answer to this question is

often not found in the Terms of Service agreement attached to these cloud services.

Client-side encryption alleviates some of these privacy concerns by limiting the number of individuals that are able to utilize the data to those possessing a secret key. However, standard symmetric encryption techniques make selective retrieval of documents in the cloud impossible because the documents can not be efficiently searched or indexed (due to the randomized nature of the encryption). Privacy enhancing technologies such as Symmetric Searchable Encryption (SSE) (e.g., [7,36,52]) have been developed to meet this need. They provide efficient means of searching over encrypted files, but do so at the cost of significant statistical information leakage that can be used to reveal much about the search queries as well as the encrypted files themselves [30]. To overcome this, researchers have employed tools such as Oblivious RAM (ORAM) [22]. Originally designed for the secure processor setting, ORAM is a cryptographic scheme that can be used to access data stored with an untrusted third-party in a way that makes the history of accesses indistinguishable from uniform random accesses (in space, not time). However, the use of ORAM introduces significant challenges to designing efficient cloud-based private search schemes due to additional bandwidth and interactivity requirements.

In this thesis, we develop schemes that begin to bridge the gap between efficient and private search in the cloud. In particular, we develop specialized ORAM schemes designed for more efficient access to tree data structures. Tree data structures are of special interest as they play a crucial role in large-scale database systems as search indexes due to their properties of being efficient for search and update. To support these new schemes, as well as foster continued improvements to ORAM and cloud-based privacy-enhancing technologies, we have also created a new software package for Python call PyORAM (Python-based Oblivious RAM) [29]. Our motivation for PyORAM is two-fold: (1) that it will enable dissemination of new ORAM technologies tailored for

cloud storage by allowing researchers to take advantage of the ability to rapidly prototype algorithms in Python, and (2) that it can serve as an easily accessible library of well-tested ORAM algorithms on top of which users can develop privacy-enhancing tools for cloud storage.

## 1.2 Related Work and Limitations

### 1.2.1 Symmetric Searchable Encryption

Song, Wagner, and Perrig first introduced Symmetric Searchable Encryption (SSE) in 2001 [42]. SSE provides a means of searching over a set of encrypted files by sending an encrypted search query to a server that runs some search algorithm and returns the set of matching encrypted documents (or document identifiers). The goal is to minimize the amount of information that the server learns about the search query while still allowing the search to be performed efficiently. These schemes are most commonly based on an *encrypted index* [7–10, 13, 20, 31, 36, 46, 52]). An *encrypted index* is an encrypted form of some data structure mapping keywords to their associated set of file identifiers. SSE schemes differ in what data structure is used, and how it is encrypted, and this affects how efficiently certain operations on this data structure can be performed (e.g., search, update). The choice of data structure also affects how much information is leaked by these operations. While some SSE schemes are highly efficient, all schemes leak significant information to the server during a search query, including decrypting portions of the encrypted index. The two leakages we concern ourselves with in this work are the *search pattern* and *access pattern*.

By the *search pattern*, we mean the history of encrypted search terms (keywords)

transmitted to the search program that runs on the storage server. The plaintext form of the search terms are not revealed, but when, and how often, search takes place for the same keyword can be readily tracked by the server due to the deterministic search tokens that must be used by these schemes. By the *access pattern*, we mean the history of accesses to the physical address space of the storage. This may include the accesses made by the search program or those made by the customer accessing the files associated with the search term.

Islam et al. demonstrated that as much as 80% of search queries can be inferred by observing the search and access pattern of an encrypted email repository [30]. More recently, Cash et al. in [6] showed a range of attacks on SSE schemes are possible based on varying levels of knowledge by the adversary about the search key or the file data. For instance, a malicious cloud storage provider could send the customer an email, knowing that the email would be added to the client's encrypted email search index. Now knowing the plaintext information of one or more files included in the client's index, the storage provider could then launch a set of attacks that reveal the plaintext form of certain customer queries. Naveed, in [35] formally defines different classes of leakages that are inherent to *any* SSE scheme, and went on to show how ORAM-based approaches can eliminate some of these leakages, but in doing so, add significant communication cost.

### 1.2.2 Oblivious RAM

An Oblivious RAM (ORAM) is a cryptographic protocol that can be used to store data with an untrusted third party in such a way as to prevent the third party from learning anything about the data from the access pattern. ORAM protocols conceal a program's **logical** access pattern by continuously shuffling and re-encrypting data as

they are accessed. The **physical** storage addresses accessed by a program remain visible to the third party party, but the ORAM hides any association between the physical access pattern and the true logical access pattern. The trivial ORAM is defined as scanning the entire storage space to protect a sequence of accesses. Although ORAM was originally designed for secure embedded processors, it has naturally been extended to the client-server model used for cloud storage.

Goldreich and Ostrovsky were the first to obtain theoretical bounds on efficiency of ORAM schemes, proving that any ORAM requires at least $O(\log N)$ overhead to conceal a logical access pattern in a storage space with $N$ items [21,37]. Along with establishing this bound, they described the first ORAM implementation that outperformed the trivial ORAM, Square Root ORAM, which achieved an amortized communication overhead of $O(\sqrt{N}\log N)$ [22]. In the decades since their seminal work, researchers have continued to develop more efficient schemes in an effort to make ORAM practical [1, 2, 4, 5, 11, 12, 14–19, 23–27, 32, 33, 38, 40, 41, 43, 44, 47, 48, 50, 51].

Stefanov et al. in [45] defined Path ORAM, which was the first ORAM scheme to achieve asymptotically optimal $O(\log N)$ communication overhead. Despite this achievement, the hidden constants associated with the communication cost are still an impediment to its use in practice [3, 35]. Moreover, Path ORAM can require significant client storage in the form of the position map. A recursive form of Path ORAM can be used to eliminate client-side storage of the position map, but it introduces additional $O(\log N)$ round complexity on every access, which significantly increases the communication cost.

To address this, Wang et al. in [49] showed how to eliminate client-side storage of the position map, with no additional communication complexity, when accessing tree data structures. The oblivious dictionary scheme in [49], when combined with Path ORAM, shows show that oblivious operations on self-balancing trees with bounded degree are

possible using $O(\log N)$ client storage and $O(\log N)$ communication overhead.

Ren et al. made further improvements at reducing the communication cost of Path ORAM with Ring ORAM [39]. Ring ORAM successfully reduces online communication overhead to $O(1)$ by introducing a constant number of additional communication rounds to each request and by leveraging server-side computations. Ring ORAM is an important milestone as it shows that the $O(\log N)$ lower bound on communication overhead for ORAM can be overcome using a different computational model (e.g., one where the storage server can perform computations). However, Bindschaedler et al. in [3] report that ORAM schemes requiring custom server-side computations are not compatible with basic cloud-storage services (e.g., Amazon S3). Also, due to high network latency, introducing even a constant number of communications rounds to a block request can have a severe negative impact on performance.

Furthermore, despite its added security, ORAM can still leak information when used in the context of searchable encryption, as shown by Naveed in [35]. When a search index is accessed using the oblivious dictionary scheme from [49], the following leakages are present: (1) the size of the search index is leaked simply by storing it on the server, and (2) the number of documents that match a query is leaked by the number of I/O requests made by the client after retrieving the matching document (even if those documents are in a separate ORAM). Padding operations can be used to partially address both leakages; however, this can result in substantial overhead such that it quickly becomes more efficient to fully scan the storage one time [35].

It is apparent that there is still significant room to improve the privacy of cloud search applications. By using ORAM, cloud search applications can greatly improve privacy guarantees, but at the cost significant communication overhead, so it is worth exploring some middle ground between ORAM and SSE. In particular, *it is worth exploring mini-*

*mal relaxations to the ORAM security model that accept certain leakages as inevitable, so that more efficient schemes can be developed that still prevent leakage of privacy-critical information.* Specifically, we are interested in schemes that do not leak the history of deterministic search tokens and that do not leak the file identifiers matching a search.

## 1.3 Research Problem and Main Insight

Our work is motivated by answering the following question:

> *Can more efficient schemes be developed for obliviously accessing external storage by making certain aspects of the client program public (which already may be inferred due to the nature of the client application)? Specifically, (i) making the program being executed public, and (ii) making the structure of certain inputs to the program public.*

We now present an illustrative example showing why the answer is affirmative. Assume that a client program is running binary search over a tree stored on an untrusted server, and that each node of the tree is stored in its own memory block. Additionally, assume that ORAM is used to retrieve nodes from storage. The following are potential leakages that occur when the binary search program is executed:

1. *The Program*: If the height of the tree is $O(\log N)$, then the number of nodes accessed by the client program will be $O(\log N)$. Thus, after accounting for the additional I/O overhead added by ORAM, an adversary (the server) might infer properties of the client program through the length of its access sequence relative to the size of storage (e.g., it operates over a data structure of height $O(\log N)$).

2. *The Search Key*: When the program is executed with two different input search keys that result in a different number of node accesses, these cases will be distinguishable to an adversary. This would occur when the tree is not perfectly balanced or when the search does not always stop at the same depth in the tree. ORAM alone does not hide the number accesses made by a program.

3. *The Tree*: The depth at which a match occurs in the tree is leaked by the number of nodes accessed by a client program (inferred after accounting for the I/O overhead of ORAM). If an adversary knows something about the search key, this would leak information about the nodes in the tree.

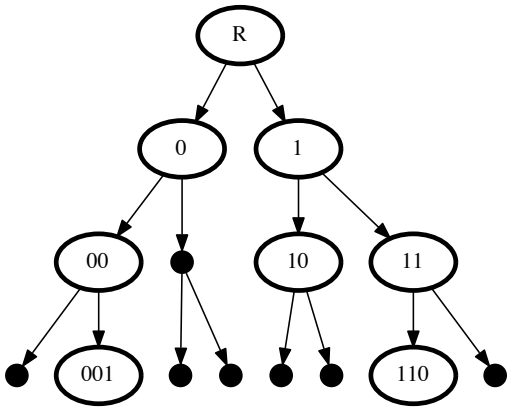
The first leakage can only be prevented by introducing significant padding operations or by using the trivial ORAM (full scan of storage). However, in SSE the client program *is* public because server is running the search program. Thus, as long as we can prevent leakage of information to the server about the input and output of the search program, it should be of no concern to hide, for instance, that we are running binary search on a tree of maximum height $h$.[1] To ensure that no information about the balance of the tree can be inferred from the amount of storage, we embed the binary tree in a perfect tree of height $h$. To prevent the second and third leakage from the list above, we can introduce no more than $O(h)$ padding operations so that the number of nodes accessed is always $(h + 1)$. We illustrate these ideas in Figure 1.1.

Each time the PaddedBinarySearch program from Figure 1.1a is executed, exactly $(h + 1)$ nodes are retrieved from external storage through separate calls to the ORAM Access function. As a result, execution of this program with any two input keys will always be indistinguishable to an adversary observing the physical access sequence.

[1] We define the height of a tree as the maximum number of edges from the root to any leaf.

```
PaddedBinarySearch(key, tree):
 1: match = tree.NIL
 2: current = Access(tree.root)
 3: while current ! = tree.NIL do
 4:     if key == current.key then
 5:         match = current
 6:     end if
 7:     if key < current.key then
 8:         current = Access(current.left)
 9:     else
10:         current = Access(current.right)
11:     end if
12: end while
13: return match
```

(a)




(b)

Figure 1.1: Motivating example.

Additionally, the physical access sequence will reveal no information about the individual nodes on the tree to an adversary with knowledge about the search key.

*The main insight is the following*: The ORAM security model ensures that *all client programs* requiring $t$ memory accesses will be indistinguishable from the point of view of the storage server. For the PaddedBinarySearch program, the number of accesses is $t = (h + 1)$. If there are $N = (2^{h+1} - 1)$ blocks of memory stored on the server (the number of nodes in a perfect binary tree of height $h$), the space of possible $t$-length access sequences has size $N^t = (2^{h+1} - 1)^{h+1} = O(2^{h^2})$. However, there are only $2^h$ possible access sequences that would ever by executed by PaddedBinarySearch. All such access sequences start with the root node, followed by one of exactly two child nodes, and this process repeats until one of the $2^h$ leaf nodes of the perfect binary tree is reached. Thus, the space of possible access sequences in this example is significantly smaller than the general case. Since the client program has been made public, remaining oblivious within its reduced space of possible of access sequences is sufficient for preventing information leakage to the server about the search key or the search tree. With this insight in mind,

we describe our contribution in further detail.

## 1.4 Our Contribution

We significantly reduce the overhead of oblivious access to tree data structures by only publicizing to the storage provider that the client program operates on a tree data structure. Recall that when using SSE schemes, this information is already made public since the search program is run by the storage provider. We focus on reducing communication overhead without introducing additional round complexity, since network latency and bandwidth are the major performance bottlenecks for ORAM. Note that this directly correlates with a reduction in client computational requirements as well, since blocks loaded from storage during an ORAM access must be fully decrypted and re-encrypted.

In this direction, we developed two schemes that we call *Oblivious Access to Trees (OAT)* and *TINY-OAT*. These schemes are able to achieve significant performance gains over standard ORAM approaches (where the client program is not publicized). In Table 1.1, we outline the comparison of our schemes with Path ORAM, which is one of the most efficient ORAM schemes to date that is compatible with current cloud-storage APIs. We analyze the transmission overhead and client storage by carefully considering the associated hidden constants in the complexity analysis. We further elaborate the desirable properties of our schemes as follows:

- **Oblivious Access to Trees (OAT)**: Our scheme leverages the observation that, for many tree-access routines (e.g., binary search), anonymous access within each level of the tree is sufficient for privacy. This approach permits OAT to achieve roughly twice the efficiency of that of Path ORAM when we consider a sequence of accesses leading from the root of the tree to some leaf node.

- **TINY-OAT**: We improve upon the OAT scheme with TINY-OAT, a specialized scheme for binary trees that is designed to excel in the case of bulk access requests where efficient caching strategies are helpful. TINY-OAT is a novel tree-based ORAM that merges the structure of the client's tree data structure with the scheme's binary storage heap. *TINY-OAT with tree-top caching achieves 7.4 times better performance than that of the state-of-the-art Path ORAM without tree-top caching*, making it an ideal choice for oblivious access to trees in cloud-storage applications. TINY-OAT achieves this improvement with a factor of $O(h)$ less client storage than OAT (hence the name).

The remainder of this thesis is organized as follows. Chapter 2 presents our preliminary concepts and definitions. Chapter 3 describes our proposed schemes in detail. In Chapter 4, we perform in-depth analysis of our proposed schemes in terms of their security and performance. We also report timing results for a real case study implemented using our software package PyORAM. Finally, Chapter 5 concludes this work.

Table 1.1: Transmission cost and client storage for compared schemes.

| **Scheme** | **Transmission Cost (online)** | **Caching Strategy** | **Client Storage** | **Cost Reduction**[a] |
|---|---|---|---|---|
| Path ORAM | $2 \cdot Z \cdot (h+1)^2$ | None | $O(h)$ | – |
| OAT | $2 \cdot Z \cdot T_{h+1}$ | | $O(h)$ | 1.9x |
| TINY-OAT | $2 \cdot Z \cdot T_{h+1}$ | | $O(h)$ | 1.9x |
| Path ORAM | $2 \cdot Z \cdot \left\lceil \frac{h+1}{2} \right\rceil \cdot (h+1)$ | Tree-Top Cache $\left\lfloor \frac{h+1}{2} \right\rfloor$ levels | $O(\sqrt{2^h})$ | 2x |
| OAT | $Z \cdot (T_{h+1} + h + 1)$ [b] | | $O(\sqrt{2^h})$ | 3.7x |
| **OAT**[*] | $2 \cdot Z \cdot T_{\left\lceil \frac{h+1}{2} \right\rceil}$ | | $O(h \cdot \sqrt{2^h})$ | **7.4x** |
| **TINY-OAT** | $2 \cdot Z \cdot T_{\left\lceil \frac{h+1}{2} \right\rceil}$ | | $O(\sqrt{2^h})$ | **7.4x** |

- *Settings*: We use the non-recursive form of Path ORAM as our baseline for comparison. In all cases, we assume the client-side position map is eliminated using the "pointer technique" discussed in Chapter 2. OAT uses Path ORAM as the underlying ORAM for each level. The bottom half of the table compares schemes when combined with the tree-top caching technique [34], in which we assume the top half of the storage heap for Path ORAM is cached on the client-side before all access requests.
- *Transmission Cost*: Values for transmission cost represent the total number of blocks transmitted between the client and the server for a client program that makes exactly $(h+1)$ requests, which define a path from the root of the tree to some leaf node. Note that values represent the online transmission cost, which excludes any offline costs, such as setting up the local tree-top cache.
- $T_n := \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (the $n$-th triangular number)
- * OAT can achieve the performance of TINY-OAT but with a factor $O(h)$ more client storage.
- [a] For a perfect binary tree with $h = 25$
- [b] A tight upper bound for the true cost $\sum_{\ell=0}^{h} 2 \cdot Z \cdot \left\lceil \frac{\ell+1}{2} \right\rceil$.

# Chapter 2: Preliminaries

## 2.1 Oblivious RAM

ORAM enables a client to access encrypted data on an untrusted server without revealing the true access pattern to the server. That is, ORAM ensures that any two client data request patterns of the same length are indistinguishable to the server. Existing ORAM schemes hide the access pattern by re-encrypting and shuffling data blocks on the server over the course of the accesses.

The client accesses the data via blocks, each $B$-bits (e.g., 128 bytes to 256 KB), with each block having a unique $\mathsf{id}$ that is only known to the client. Any read or write request made to a block by the client is called a *virtual* I/O request. Given a sequence of virtual I/O requests, an ORAM produces a sequence of requests to the server that fulfill these requests. We refer to the access sequence seen by the server as the *physical* access sequence. Each time blocks are downloaded from the server, they are re-encrypted using a randomized symmetric encryption scheme, so that they never appear the same when returned to the server.

An ORAM scheme is defined in terms of a function $\mathsf{Access}(\mathsf{op}, \mathsf{id}, \mathsf{data})$, where $\mathsf{op}$ denotes the operation ($\mathsf{read}$ or $\mathsf{write}$), $\mathsf{id}$ identifies the block on which to perform the operation, and $\mathsf{data}$ defines the new information to store in the block when the operation is a $\mathsf{write}$.

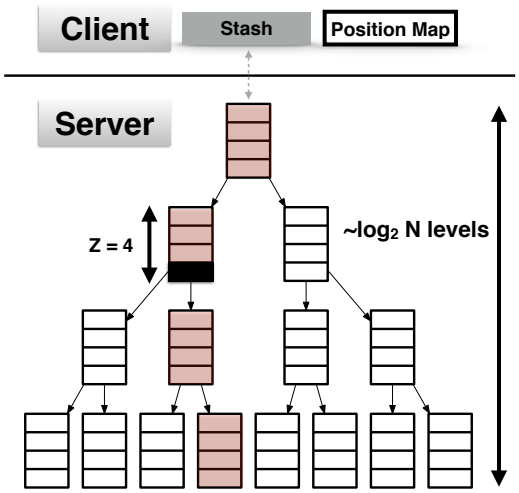



Figure 2.1: Major components of Path ORAM.

## 2.2 Path ORAM

Stefanov et al. in [45] proposed Path ORAM, which is a tree-based ORAM with asymptotically optimal communication overhead. Figure 2.1 shows an architectural overview of Path ORAM. The server storage is structured as a binary heap of height $h \approx \log N$, where $N$ is the number of memory blocks. Each node in the heap is referred to as a bucket, which consists of $Z$ memory blocks. The Path ORAM scheme requires that the bucket size parameter $Z$ be sufficiently large ($Z \geq 4$) to prevent excessive client storage [34, 45]. The client maintains a position map with entries $(\mathsf{id}, \mathsf{pos})$ for every block, where $\mathsf{id}$ is the block identifier and $\mathsf{pos}$ is one of the $2^h$ leaf bucket identifiers in the storage heap.

The client accesses a block (black) by downloading all buckets along the path (shaded grey) from the root of the heap to the leaf bucket assigned to that block in the position map. The $Z \cdot (h + 1)$ blocks along this path are temporarily placed in the client's local storage, which is referred to as the *stash*. Roughly half of the blocks downloaded with

the path will be dummy blocks, marking empty positions in the buckets. The requested block is then assigned a new random address (leaf bucket) in the heap and the local position map is updated. The same path is then re-populated with blocks from the client stash and transmitted back to the server. Thus, each virtual I/O access requires $2 \cdot Z \cdot (h + 1)$ physical block transmissions between the client and the server.

The process of moving blocks from the stash to some position in the heap path that will be transmitted back to the server is called *eviction*. Path ORAM uses a greedy eviction procedure whereby blocks are pushed as far down in the current path as possible (respecting bucket capacity constraints), such that they still lie along the path to their assigned leaf bucket. For instance, if a block from the stash has been assigned one of the leaf buckets below the right child of the root bucket, and the path to be evicted goes through the left child of the root bucket, then that block can be placed no lower than the root bucket during eviction. If the bucket size $Z$ is too small (e.g., less than 4), buckets near the root will tend to become congested, and cause the stash to grow large with blocks that fail to evict.

Path ORAM offers a client storage and communication overhead trade-off. The client can achieve $O(\log N)$ asymptotically optimal communication overhead by storing the $O(N \log N)$ size position map locally. The size of the position map is due to the fact that it stores $N$ entries, each of which is size $\log N$ (the number of bits necessary to represent a leaf bucket id). For clients with moderate storage resources, this is often reasonable. For instance, for 1 TB of storage with a block size of 4 KB, the position map requires $\sim$1 GB of local storage. Alternatively, the client can use a recursive variant of Path ORAM to achieve $O(h)$ storage overhead, but requiring $O(h)$ additional rounds of communication per I/O request. Transmission cost can be reduced using a method known as *tree-top caching* [34]. The idea is to store the top $t$ levels of the binary heap on

the client prior to running a set of access requests. This reduces the number of buckets that must be transmitted between the client and server each time an access request is made (online cost) by $2 \cdot Z \cdot t$, but increases the client storage by $Z \cdot (2^{t+1} - 1)$ blocks. Caching the top half of the heap (the first $\lfloor \frac{h+1}{2} \rfloor$ levels) reduces the online cost by a factor of $\sim$2, and requires $O(\sqrt{N}) = O(2^{h/2})$ blocks of client storage. For the same 1 TB storage example above, caching the top half of the storage heap requires $\sim$65 MB of local client storage.

## 2.3 Oblivious Data Structures and the Pointer Technique

Wang et al. proposed in [49] the "pointer technique" to eliminate bulk storage of the position map on the client when accessing tree data structures. It works by breaking the position map into smaller pieces, and storing these pieces within the ORAM blocks where they are needed. For tree data structures with bounded degree (the maximum number of children per-node is $O(1)$), each node's storage block is augmented with $O(1)$ additional slots that hold the ORAM position of the block itself, along with the positions and identifiers of any child node blocks. An ORAM block is then defined as $\mathsf{block} := (\mathsf{id}, \mathsf{data}, \mathsf{pos}, \mathsf{childmap})$, where $\mathsf{id}$ is the block identifier, $\mathsf{data}$ is client data, $\mathsf{pos}$ is the block's ORAM position, and $\mathsf{childmap}$ is a list of $(\mathsf{id}_c, \mathsf{pos}_c)$ entries for each child $c$ of the node. To ensure that the $\mathsf{childmap}$ is up to date, a child block must only be accessible through at most one parent block. Any blocks that do not have a parent (e.g., the root node of a tree) must remain in a position map stored with the client.

In addition to the "pointer technique", Wang et al. in [49] also discuss a stronger notion of private access to self-balancing tree data structures that uses a bounded number padding operations to hide which tree routine is being called by the client. For instance,

their oblivious dictionary scheme, which is based on an AVL tree, uses $O(h)$ padding operations to ensure that search, node insertion, and node removal all require the same number of accesses to storage, and therefore, are indistinguishable to the server due to the underlying ORAM that is used. Although padding operations can be used in a similar way with our schemes, we consider our work orthogonal to theirs, as we are using a slightly weaker notion of oblivious access to a tree data structure by publicizing the level of the tree from which we access a node. In future work, we will consider how to combine our schemes with the oblivious dictionary scheme in [49].

# Chapter 3: Proposed Schemes

We propose two new ORAM schemes that achieve high efficiency by publicizing that the client operates on a tree data structure (which already may be known or inferred by the server as discussed in Section 1.3). Specifically, we rely on the simple, yet powerful, observation that it is sufficient to be oblivious *within the same level of a tree* for various client programs (e.g., binary-search). Therefore, one can partition the storage space by each tree level, and then use individual ORAMs to access each space. This core idea exactly describes the OAT scheme. We specialize this strategy for the case of binary trees by combining separate Path ORAM instances using a heap unification strategy, which results in a completely new ORAM scheme, TINY-OAT, that is highly efficient for tree access and excels in the case of caching strategies such as tree-top caching.

## 3.1 Oblivious Access to Trees (OAT)

Our first scheme *Oblivious Access to Trees (OAT)* is based on a simple but highly effective partitioning strategy, which achieves oblivious access within each level of the tree. This provides several desirable properties: *(i)* The overhead associated with obliviously accessing a node scales with the depth at which the node lives in the tree. This is because the overhead of any ORAM scheme is inherently a function of the number of blocks stored in that ORAM. When a single ORAM is used, the same lower bounds on overhead apply whether or not one accesses the root node or a leaf node. *(ii)* OAT easily generalizes to $k$-ary trees as well as trees with non-uniform degree. *(iii)* Any type of

Table 3.1: Notation.

| | |
|---|---|
| $N$ | Number nodes in the tree data structure |
| $h$ | Height of the tree data structure |
| $\mathcal{L}(\mathsf{id})$ | Level for node (block) identified by $\mathsf{id}$ in the tree data structure |
| $Z$ | Capacity of each bucket (in blocks) in the storage heap |
| $\mathcal{P}(b)$ | Path from bucket $b$ to root bucket in the storage heap |
| $\mathcal{P}(b, \ell)$ | Bucket at level $\ell$ along the path $\mathcal{P}(b)$ |
| $S$ | Client's local stash |
| $\mathsf{position}$ | Client's local position map |
| $b := \mathsf{position}[\mathsf{id}]$ | Node (block) identified by $\mathsf{id}$ is currently associated with bucket $b$, i.e., it resides somewhere along $\mathcal{P}(x)$ or in the stash. |

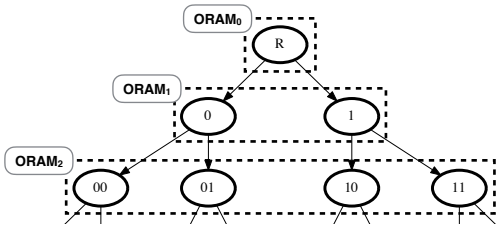



Figure 3.1: Illustration of OAT scheme for binary trees.

ORAM can be used to store the nodes at each level, and each ORAM can used its own distinct block size, which is optimized for the nodes in that level.

In our construction, we assume each node is stored in a separate block of size $B$. This can be extended to nodes that require multiple blocks for storage, but we assume all nodes require the same number of blocks to avoid discussion of padding operations while describing our scheme. We also assume that the level in which a node belongs in the tree can be inferred from its block $\mathsf{id}$. For binary trees, the standard binary heap layout achieves this. The root node is assigned $\mathsf{id} = 0$, and the left and right child of every node are assigned recursively as $2 \cdot \mathsf{id} + 1$ and $2 \cdot \mathsf{id} + 2$, respectively. A function that computes the level associated with a given $\mathsf{id}$ is then defined as $\lfloor \log_2(\mathsf{id} + 1) \rfloor$. In general, we denote this level-computing function as $\mathcal{L}$. In practice, this level computing function is not required as the $\mathsf{Access}$ function can be defined with an additional input specifying the tree level. We exclude this input from our definitions to make our schemes more

readily comparable to standard ORAM definitions. Table 3.1 summarizes our notation.

The OAT scheme is setup by partitioning the nodes of the tree of height $h$ by level. For each of these $(h+1)$ partitions, a separate ORAM is initialized with sufficient storage for all nodes in that level. Figure 3.1 illustrates this for a perfect binary tree. The $\mathsf{Access}$ function for OAT is defined as

$$\mathsf{Access}(\mathsf{op}, \mathsf{id}, \mathsf{data}) := \mathrm{ORAM}_{\mathcal{L}(\mathsf{id})}.\mathsf{Access}(\mathsf{op}, \mathsf{id}, \mathsf{data}), \tag{3.1}$$

where $\mathrm{ORAM}_{\mathcal{L}(\mathsf{id})}$ represents the ORAM storing all nodes in level $\mathcal{L}(\mathsf{id})$ of the tree.

## 3.2 TINY-OAT

We now present TINY-OAT, which significantly improves OAT for the case of accessing binary trees. TINY-OAT reduces the server storage of OAT, achieving the same storage of a single Path ORAM instance. Moreover, TINY-OAT increases the effectiveness of caching operations, like tree-top caching, that speed up bulk access requests. For instance, tree-top caching the top half of the storage heap is *3-4 times more effective at reducing transmission cost* with TINY-OAT than with Path ORAM.

The TINY-OAT construction can be described as unifying the $(h+1)$ storage heaps and stashes used by OAT when Path ORAM is used to store the nodes in each tree level. This unification process is defined as follows: First, merge the $(h+1)$ stashes into a single stash $S = \cup_\ell S_\ell$. Second, construct a single binary heap of height $h$ using bucket size $Z$. Finally, for each level $0 \leq \ell \leq h$, let the bucket at position $0 \leq j \leq 2^\ell - 1$ within that level in the new heap contain all non-empty blocks from buckets at the same position in that level of the other heaps (if that level exists). If a bucket is full, place blocks in the

**TINY-OAT.Access**$(\mathsf{op}, \mathsf{id}, \mathsf{data}^*)$: [2]

1: $b \leftarrow \mathsf{position}[\mathsf{id}]$

*Assign a random bucket within the same level*

2: $\mathsf{position}[\mathsf{id}] \leftarrow \text{Random}([2^{\mathcal{L}(\mathsf{id})} - 1, 2 \cdot (2^{\mathcal{L}(\mathsf{id})} - 1)])$

*Read buckets only up to level* $\mathcal{L}(\mathsf{id})$

3: **for** $\ell \in \{0, 1, \ldots, \mathcal{L}(\mathsf{id})\}$ **do**

4: $\quad S \leftarrow S \cup \text{ReadBucket}(\mathcal{P}(b, \ell))$

5: **end for**

6: data $\leftarrow$ Read block $\mathsf{id}$ from $S$

7: **if** $\mathsf{op} = \mathsf{write}$ **then**

8: $\quad S \leftarrow (S - \{(\mathsf{id}, \mathsf{data})\}) \cup \{(\mathsf{id}, \mathsf{data}^*)\}$

9: **end if**

*Evict blocks no lower than their assigned bucket*

10: **for** $\ell \in \{\mathcal{L}(\mathsf{id}), \mathcal{L}(\mathsf{id}) - 1, \ldots, 0\}$ **do**

11: $\quad S' \leftarrow \{(\mathsf{id}', \mathsf{data}') \in S \ : \mathcal{P}(b, \ell) = \mathcal{P}(\mathsf{position}[\mathsf{id}'], \ell)\}$

12: $\quad S' \leftarrow$ Select $\min(|S'|, Z)$ blocks from $S'$

13: $\quad S \leftarrow S - S'$

14: $\quad \text{WriteBucket}(P(b, \ell), S')$

15: **end for**

16: **return** data

Figure 3.2: Access protocol for TINY-OAT.

parent bucket (or the stash if at the root bucket).

The Access function for TINY-OAT is shown in Figure 3.2. To simplify the notation, we express TINY-OAT using an explicit client-side position map. However, when analyzing this scheme we always assume this is eliminated using the "pointer technique" that has been previously described.

[2] We emphasize that lines 2, 3, and 10 are where this algorithm differs from Path ORAM.

# Chapter 4: Analysis and Comparison

## 4.1 Security Analysis

Our security analysis, as in the case of Path ORAM [45], is concise, since the security of OAT schemes are evident from their building block ORAM. We assume that the server is untrusted, and the client is trusted, including the client's processor, memory, and disk.

**Definition 1** (Oblivious RAM)**.** Denote a data request sequence of length $M$ as

$$\vec{y} := ((\mathsf{op}_1, \mathsf{id}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_M, \mathsf{id}_M, \mathsf{data}_M)),$$

where each $\mathsf{op}_i$ denotes a $\mathsf{read}(\mathsf{id}_i)$ or a $\mathsf{write}(\mathsf{id}_i, \mathsf{data})$ operation. Let $A(\vec{y})$ denote the sequence of accesses made to the server that satisfies the user data request sequence $\vec{y}$. An ORAM construction is secure if: (1) for any two data request sequences $\vec{y}$ and $\vec{z}$ of the same length, the access patterns $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable to an observer, and (2) it returns on input $\vec{y}$, data that is consistent with the original data request sequence with probability $\geq 1 - \mathsf{negl}(|\vec{y}|)$. That is, the ORAM fails to complete the data request with only a negligible probability.

**Definition 2** (Oblivious Tree Access)**.** Let $T$ be a tree data structure of height $h$. Let $T_\ell$ be the set of nodes at level $0 \leq \ell \leq h$ in the tree. A construction providing *oblivious tree access* to nodes $u \in T$ leaks no information from the access pattern beyond the level of the tree $T_\ell$ where each access takes place.

**Theorem 1.** *OAT schemes satisfy Definition 2 when, for each level $\ell$, all tree nodes within that level $u \in T_\ell$ require the same number of virtual I/O requests to retrieve them from storage.*

*Proof.* OAT uses a secure ORAM that satisfies Definition 1 to access each level of the tree. Thus, as long as a node accessed within level $\ell$ is not distinguishable from any other node within that level through the number of access requests, the ORAM will provide the necessary indistinguishability by definition. Thus, OAT satisfies Definition 2, and this trivially establishes the same property for TINY-OAT.[3] □

**Comparison with Oblivious Data Structures.** Oblivious data structures (ODS) is a stronger form of security than ORAM (and Oblivious Tree Access). It uses additional padding operations to prevent leaking what kind of operation takes places on a data structure. We have no such padding operations built into the definition of our scheme. We note, however, that inclusion of such padding operations (e.g., the PADDED-BINARYSEARCH example) can be used to prevent additional leakages about inputs for certain client programs.

## 4.2 Performance Analysis

We analyze the performance of our schemes for the case of perfect binary trees of height $h$. Our access sequence follows a path from the root ($\ell = 0$) to some node at the leaf level ($\ell = h$), which requires $(h + 1)$ virtual I/O requests (i.e., one request to each level of the tree).

We analyze transmission cost and client storage for this access sequence in three

[3]Stash size analysis for TINY-OAT is necessary to ensure that this definition is satisfied without $O(N)$ client storage. We provide strong empirical evidence of this in Section 4.3

different settings. In the first setting, we use a single instance of Path ORAM to access tree nodes. This serves as a baseline for measuring the performance of the OAT and TINY-OAT schemes. In the second setting, we use the OAT scheme, where the nodes in each level of the tree are partitioned across $(h+1)$ separate instances of Path ORAM. In the third setting, we use the TINY-OAT scheme. In each setting, we assume the "pointer technique" is used to eliminate client-side storage of any position map. For each of the schemes, we first perform the analysis assuming no caching strategy is used, and then we repeat the analysis for the case of tree-top caching the first $\left\lfloor \frac{h+1}{2} \right\rfloor$ levels. In this case, we compute the transmission cost in terms of the online cost. That is, we ignore the cost of downloading the cached memory prior to performing the sequence of accesses.

These results are summarized in Table 1.1. Figure 4.1 illustrates the trend lines for transmission cost in $\log_2$ scale as a function $h$ for the various schemes, which we derive in the sections that follow. For each scheme, the cost without tree-top caching is denoted in black, while the online cost when tree-top caching half of the levels is denoted with grey. For reference, we also include in Figure 4.1, the cost for the trivial ORAM using a dashed line ($O(2^h)$), and the cost for the non-oblivious case using a solid line ($O(h)$).

### 4.2.1 Path ORAM

All tree nodes are stored in a single Path ORAM with height $h$ and bucket size $Z$. The total server storage is then $Z \cdot (2^{h+1} - 1)$ blocks. Since there is no position map, the worst case client-side storage occurs after a heap path is downloaded from the server. The number of blocks in a heap path is $Z \cdot (h+1)$. Thus, the worst-case client storage is $O(h)$. There are $(h+1)$ virtual I/O requests in the access sequence we consider, and the number of blocks transmitted between the client and the server for each request is

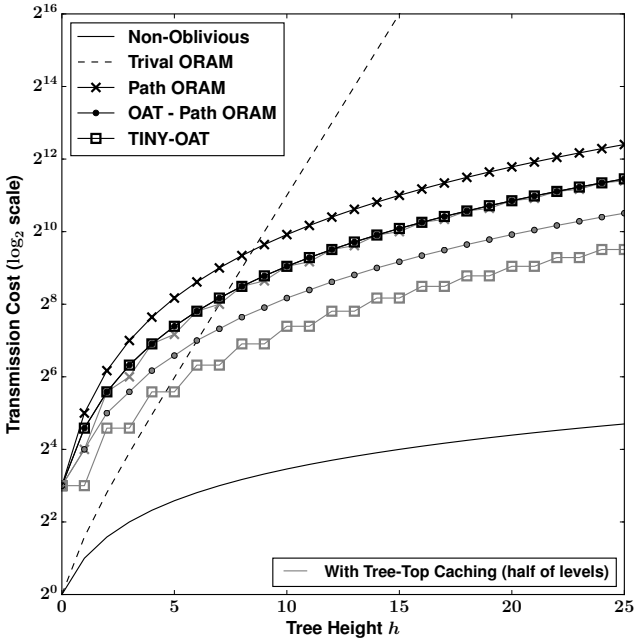



Figure 4.1: Online transmission cost with and without tree-top caching.

$2 \cdot Z \cdot (h+1)$. Therefore, the total transmission cost $C(h)$ is

$$C(h) = 2 \cdot Z \cdot (h+1)^2. \tag{4.1}$$

Suppose the client prefetches the first $\lfloor \frac{h+1}{2} \rfloor$ levels of the storage heap. The total number of blocks that the client must store is $Z \cdot (2^{\lfloor \frac{h+1}{2} \rfloor} - 1) = O(\sqrt{2^h})$. For each virtual request, the client must download buckets in the remaining $\lceil \frac{h+1}{2} \rceil$ levels from the storage heap on the server (and transmit them back). Thus, the online transmission cost for this amount of tree-top caching is

$$C(h) = 2 \cdot Z \cdot \left\lceil \frac{h+1}{2} \right\rceil \cdot (h+1). \tag{4.2}$$

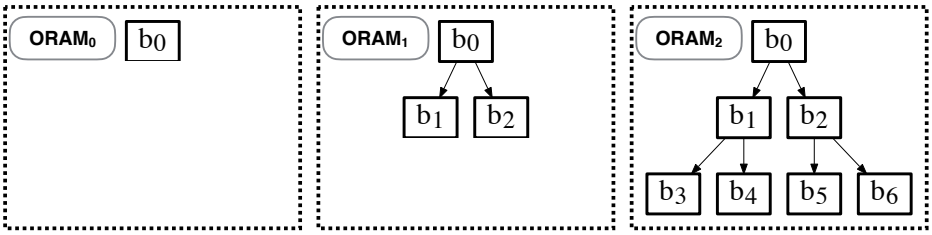



Figure 4.2: Illustration of storage heaps used by OAT scheme.

### 4.2.2 OAT

We access the tree nodes using the OAT scheme, configured with a separate Path ORAM for each of the $(h+1)$ ORAMs. To simplify the analysis, we assume that the Path ORAM at level $\ell$ (storing $2^\ell$ tree nodes) uses a storage heap with height $\ell$. That is, the Path ORAM at level $\ell$ contains $2^{\ell+1} - 1$ buckets of size $Z$. Figure 4.2 illustrates this for the first three levels of storage. Note that in practice the height of the memory heap used by Path ORAM for level $\ell$ can be reduced to $\ell - 1$ when using the recommended settings for bucket size. Stefanov et al. in [45] used the formula $L = \lceil \log_2(N) \rceil - 1$ to compute the height of the storage heap for $N$ real blocks of client storage. Additionally, for at least the first few ($\geq 2$) levels of the tree, oblivious access to nodes within these levels is more efficiently achieved by downloading all of the nodes with each access (trivial ORAM) as opposed to using Path ORAM. For simplicity, we do not consider these minor optimizations when analyzing the performance of our scheme. As Figure 4.1 shows, OAT achieves, without tree-top caching, equivalent performance to that of the Path ORAM combined with tree-top caching the top half of the storage heap.

The total server storage used by OAT is computed as:

$$\begin{aligned}\sum_{\ell=0}^{h} Z \cdot (2^{\ell+1} - 1) &= Z \cdot \left[2^{h+1}(1 + \frac{1}{2} + \cdots + \frac{1}{2^h}) - (h+1)\right] \\ &\leq 2 \cdot Z \cdot (2^{h+1} - 1)\end{aligned} \tag{4.3}$$

That is, the total server storage used by OAT is at most twice that of a single Path ORAM instance.

As in the previous setting, we assume the client-side position map is eliminated using the "pointer technique". It is worth mentioning that our scheme uses a slight generalization of this technique in that the position map of each ORAM is stored within the blocks of a *different* ORAM. That is, the position map for $\text{ORAM}_\ell$ is stored within the respective parent node blocks in $\text{ORAM}_{\ell-1}$, where the root node is stored at $\ell = 0$ and its address is always kept with the client. Thus, the client stores $(\mathsf{id}, \mathsf{pos})$ for the root node along with $(h+1)$ separate Path ORAM stashes. If using the recommended bucket size for each Path ORAM, we may assume that, with high probability, each of these $(h+1)$ stashes has $O(1)$ blocks inside of it after eviction. Note that eviction for $\text{ORAM}_\ell$ takes place before loading the path for $\text{ORAM}_{\ell+1}$, and that $\text{ORAM}_h$ has the most buckets in its path $(Z \cdot (h+1))$. Thus, worst-case client storage is $O(h)$.

The total transmission cost for the request sequence is found by summing the transmission costs from a single access to each of the $(h+1)$ Path ORAMs used to store the tree levels. Since the Path ORAM storing tree nodes in $T_\ell$ has height $\ell$, its access cost is $2 \cdot Z \cdot (\ell + 1)$. Thus, we can write the total cost as

$$C(h) = \sum_{\ell=0}^{h} 2 \cdot Z \cdot (\ell + 1) = 2 \cdot Z \cdot T_{h+1}, \tag{4.4}$$

where $T_{h+1}$ is the $(h+1)$-th triangular number.

For the case of tree-top caching, we interpret this as meaning that the top $\lfloor \frac{\ell+1}{2} \rfloor$ levels are cached for ORAM$_\ell$. The total number of blocks stored on the client is

$$\begin{aligned} \sum_{\ell=0}^{h} Z \cdot (2^{\lfloor \frac{\ell+1}{2} \rfloor} - 1) &\leq Z \cdot \left[ 2^{\frac{h+1}{2}} (1 + \frac{1}{2} + \cdots + \frac{1}{2^h}) - (h+1) \right] \\ &\leq 2 \cdot Z \cdot (2^{\frac{h+1}{2}} - 1). \end{aligned} \tag{4.5}$$

Thus, client-side storage is $O(\sqrt{2^h})$, and at most twice that required by Path ORAM.

The total transmission cost for the request sequence of interest is computed by summing the individual transmission costs of the ORAMs used for each level. For the ORAM of level $\ell$, there are $\lceil \frac{\ell+1}{2} \rceil$ remaining heap levels whose buckets are not in the local client cache. Thus, we can write total transmission cost as

$$\begin{aligned} C(h) &= \sum_{\ell=0}^{h} 2 \cdot Z \cdot \left\lceil \frac{\ell+1}{2} \right\rceil \\ &= 2 \cdot Z \sum_{j=1}^{h+1} \left\lceil \frac{j}{2} \right\rceil \\ &= 2 \cdot Z \sum_{j=1}^{h+1} \left\lfloor \frac{j+1}{2} \right\rfloor \\ &\leq 2 \cdot Z \sum_{j=1}^{h+1} \left( \frac{j+1}{2} \right) = Z \cdot (T_{h+1} + h + 1). \end{aligned} \tag{4.6}$$

### 4.2.3 TINY-OAT

We access the tree nodes using the TINY-OAT scheme. As described in Section 3.2, the storage format used by TINY-OAT is equivalent to that of Path ORAM. Thus, the server storage is equivalent to the Path ORAM case. As in the previous settings, we

assume the client-side position map is eliminated. In Section 4.3, we will study the stash size behavior of TINY-OAT. For now, we will assume that its stash size behavior is comparable to Path ORAM when a sufficiently large ($Z \geq 4$) bucket size is used. Thus, let us assume that the worst-case client storage for this setting is $O(h)$, in the case where no caching strategy is used. Derivation of the transmission cost for the access sequence we are considering is identical to that for the OAT setting. The only difference is that the $(h+1)$ access requests are made *to the same* storage heap. A node request within level $\ell$ in the tree requires transmitting buckets at levels $0, 1, \ldots, \ell$ of the storage heap.

As mentioned, an important characteristic of the TINY-OAT scheme is its increased performance when combined with caching strategies like tree-top caching. We now explain why this is the case. First, note that the client storage when caching the top half of the storage heap in the TINY-OAT setting is $O(\sqrt{2^h})$ and identical to the Path ORAM setting. Recall, however, that the modified eviction strategy used by TINY-OAT prevents nodes at level $\ell$ of the tree from being placed below level $\ell$ in the storage heap. As a result, tree-top caching up to level $\ell$ of the storage heap guarantees that all nodes from level 0 to level $\ell$ of the tree can be found in the client's local cache (or stash). Thus, when tree-top caching $\left\lfloor \frac{h+1}{2} \right\rfloor$ levels of the storage heap locally, only nodes in the remaining $\left\lceil \frac{h+1}{2} \right\rceil$ levels of the tree must be retrieved from storage, and the cost for each of those nodes is reduced by $\left\lfloor \frac{h+1}{2} \right\rfloor$ buckets. Thus, the online transmission is

$$C(h) = \sum_{j=1}^{\left\lceil \frac{h+1}{2} \right\rceil} 2 \cdot Z \cdot j = 2 \cdot Z \cdot T_{\left\lceil \frac{h+1}{2} \right\rceil}. \tag{4.7}$$

## 4.3 Stability Analysis of TINY-OAT

The eviction and addressing procedures used by TINY-OAT are similar to those used by Path ORAM. However, instead of assigning all tree node blocks a random leaf-bucket in the storage heap, we assign them a random bucket that is at the same depth in the heap as they are in the tree. The stash eviction procedure greedily pushes blocks as far down the current path as possible *without* allowing a block to move below the heap bucket it is assigned to. That is, internal nodes in the tree data structure will never reside below their corresponding internal heap bucket. Thus, a node at level $\ell$ in the tree will never reside in a bucket deeper than level $\ell$ in the storage heap.

It is important to consider the effect that this modified addressing and eviction behavior can have on the average bucket usage within levels of the heap as well as the expected stash size after each eviction. One might consider applying similar modifications to the original Path ORAM algorithm in the case of random access (not tree access). That is, allow the block position `pos` to be a random bucket anywhere in the heap, and do not allow a block to move into any level below its assigned bucket during eviction. Intuitively, one would expect the following: (i) an average access cost of less than $(h+1)$ buckets, because not all accesses need to download a full path to some leaf bucket, (ii) an increase in bucket usage near the top of the heap, and (iii) a possible increase in the average client stash size after eviction. Investigating point (i), a simple calculation shows that these modifications do little in terms of reducing the average transmission cost. This is seen by computing the average number of buckets downloaded per random access:

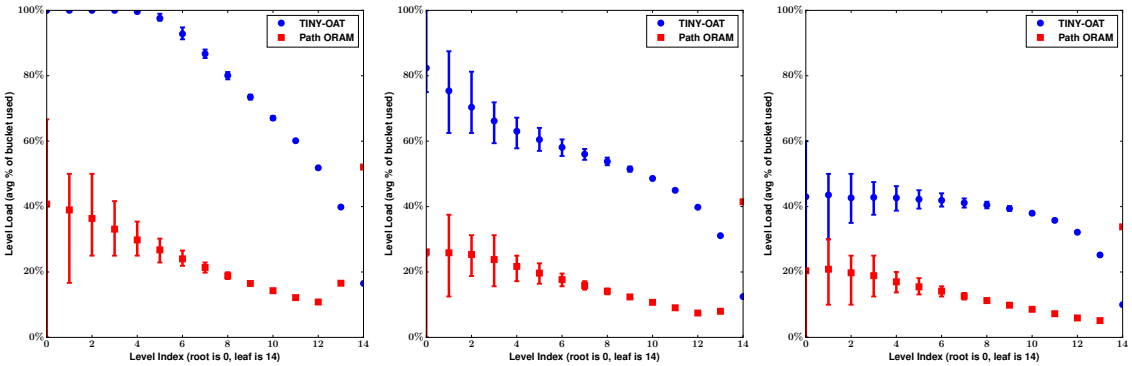
$$\sum_{\ell=0}^{h}(\ell+1)\frac{2^{\ell}}{2^{h+1}-1} = \frac{h\cdot 2^{h+1}+1}{2^{h+1}-1} \geq h. \tag{4.8}$$

That is, the average number of buckets downloaded per random access is less than $(h+1)$ but at least $h$. The analysis in Section 4.2 explains why these custom addressing and eviction procedures are beneficial in the tree access model.

We show empirically, using simulations, that the expected stash size for TINY-OAT behaves similar to that of Path ORAM with a bucket size of $Z \geq 5$. At this bucket size, average bucket load within the TINY-OAT storage heap appears to approach that of the stationary distribution when using an infinitely large bucket size. Stash size behavior for the $Z = 4$ case is slightly worse than Path ORAM, but still reasonable for practical tree sizes. Note that such empirical analysis was used to analyze Path ORAM and its variants [34, 39, 45].

Our first empirical study runs experiments on storage heap of height $h = 14$ storing $N = 2^{15} - 1$ blocks (one block per bucket). We run the experiments with different settings for the bucket size $Z$ to observe its effect on the stash size and bucket usage. We treat ORAM blocks as nodes in a perfect binary tree of $h = 14$ (the same height as the storage heap). We insert nodes into storage in breadth-first order via the Access function, which is followed by a series of $(h + 1)$-length access requests, each of which consists of accessing a path of nodes from the root to a random leaf node in the binary tree. A single round for an experiment consists of executing $2^{14}$ random root-to-leaf access sequences as described.

Figures 4.3 - 4.4 show the results of these experiments for Path ORAM and TINY-OAT. The results were generated by first running 1000 warmup rounds (as defined above), and then collecting statistics over 1000 test rounds. Figure 4.3 shows, for each level of the storage heap (horizontal axis), the average percentage of blocks that were non-empty for buckets within that level (vertical axis). In each sub-figure, markers represent the mean with error bars representing the upper and lower quartiles centered around the median.




(a) $Z = 3$ (b) $Z = 4$ (c) $Z = 5$

Figure 4.3: Average bucket load for TINY-OAT and Path ORAM.

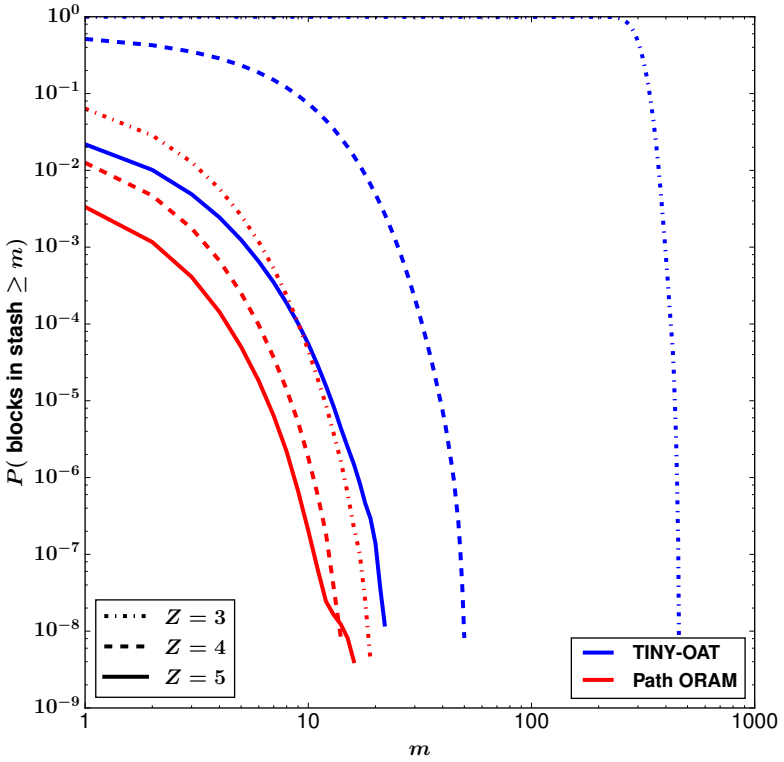
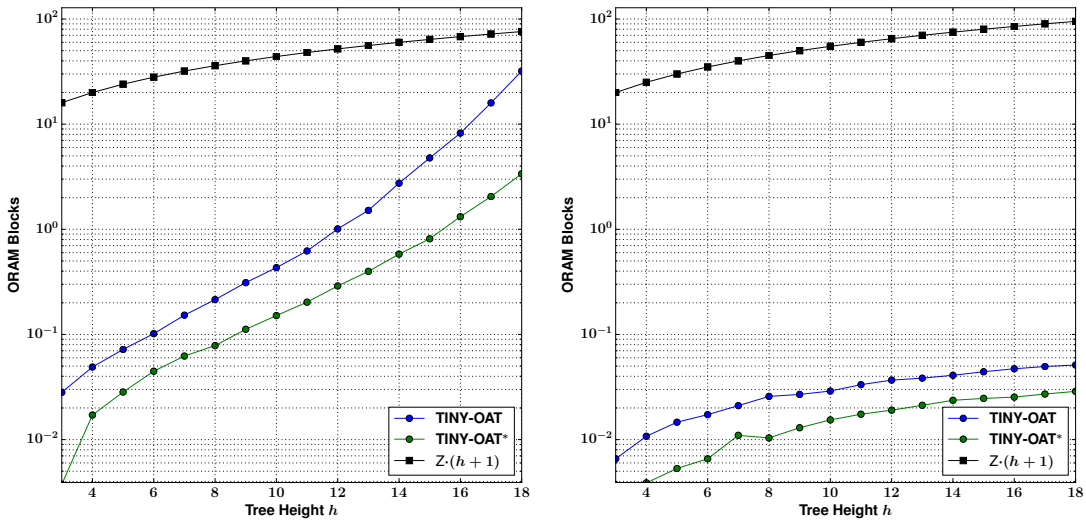



Figure 4.4: Probability of stash size exceeding $m$ for TINY-OAT and Path ORAM.




(a) $Z = 4$ (b) $Z = 5$

Figure 4.5: Average stash size for TINY-OAT

Figure 4.5b shows that for TINY-OAT (blue), with a bucket size $Z = 5$, buckets near the root of the storage heap contain on average two non-empty blocks (one more than the average number of blocks assigned to them). At the same bucket size, Path ORAM (red) contains an average of one non-empty block per bucket near the root, with a much larger concentration of buckets at the leaf level. Figure 4.4 shows that with a bucket size $Z = 4$, the probability of the stash size exceeding $O(h)$ for TINY-OAT appears to diminish quickly. Of additional interest is the behavior of Path ORAM in the tree access setting. These results suggest that smaller bucket sizes might be appropriate for accessing tree data structures through Path ORAM. Prior to this work, Path ORAM had only been analyzed in the random-access setting.

In the next set of experiments, we expand on the previous analysis by observing the average stash size of TINY-OAT as a function of the tree height $h$. Figure 4.5 shows these results. In this figure, the vertical axis counts ORAM blocks ($log_{10}$-scale) and the

horizontal axis denotes the tree height $h$. The blue sequence of points represents the average stash size for TINY-OAT as a function of the tree height $h$. The figure on the left was generated using a bucket size $Z = 4$ and the figure on the right used a bucket size $Z = 5$. For the case of $Z = 5$, there is clearly a very weak dependence on tree height, with the average stash size remaining close to zero for the entire sequence of points. However, for $Z = 4$, the average stash size for TINY-OAT shows an exponential dependence on tree height. We emphasize, however, that the average stash size for TINY-OAT is no larger than the number of ORAM blocks that would be downloaded when accessing a leaf node (i.e., that standard Path ORAM access cost) for trees up to a height of $\sim$20. For reference, we show this cost using the black sequence of points ($Z \cdot (h + 1)$).

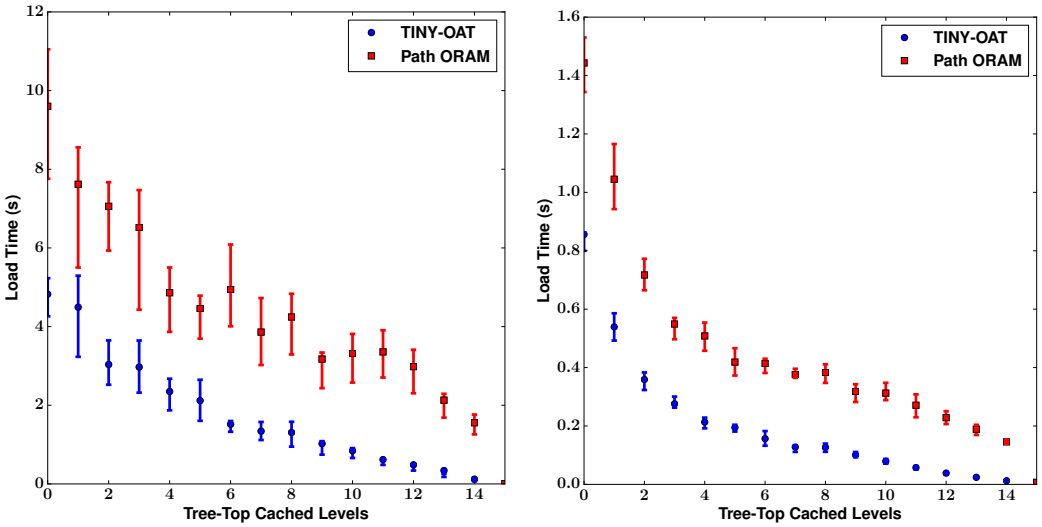
To overcome this growth in average stash size for larger tree sizes, we consider a modified form of TINY-OAT that reduces bucket usage near the root of the storage heap. Stash size results for this modified form are denoted by the orange sequence in Figure 4.5 and given the label TINY-OAT$^*$. In this modification, each time we access a node at level $\ell$, we access the storage heap as if we are accessing a node at level $\ell + 1$ (except at the last level). That is, we access all buckets along the path to the assigned heap bucket, and extend this path by a single bucket, chosen from the left or right child of the assigned bucket at random. For this case of $Z = 4$, this simple modification to TINY-OAT reduces the average stash size by a factor of 10, with an additive increase to the access cost of $2 \cdot Z \cdot h$ blocks.

## 4.4 Case Study

In the following case study, we test real implementations of TINY-OAT and Path ORAM on top of two different cloud service architectures: (1) Amazon Simple Storage Service

(S3), and (2) VM-based storage via SSH File Transfer Protocol (e.g., Amazon EC2, Microsoft Azure, Google Compute Engine). S3 storage is the more competitive option price-wise; however, requests for objects in storage often have a much higher latency than requests using storage type (2), although this can vary based on the relative physical locations of the client and server as well as network access speeds. Our test code is available for download [28]. Experiments were performed on a 64bit Macbook Pro laptop with a 2.6 GHz Intel Core i7 processor and 16 GB of RAM. Our test code was implemented in Python and made heavy of the open source Python software package PyORAM, which we have released as part of this work [29].

Figure 4.6 shows the results of our case study. In these tests, a storage space of 131 MB was created using a 4 KB block size. Using a standard bucket size parameter (Z=4) and heap height setting (h=15), this results in a total storage size of 1 GB on the server. Tests were conducted from Oregon State University over campus WiFi. For the S3 the case, we used a server in the US-West region (Oregon), which had an average ping time of 40 ms at the time these tests were run. For the SFTP case, we used a VM instance created with the Google Cloud Platform in the us-central1-b region, which had an average ping time of 65 ms at the time these tests were run. For both Path ORAM and TINY-OAT, after the storage space was setup, the timing experiment consisted of performing 25 random root-to-leaf access sequences (the same sequence analyzed in Section 4.2). The experiment was then repeated for tree-top caching from as little as one level, all the way up to 15 levels. In Figure 4.6, the blue and red markers indicate the average amount of time it took to read a single root-to-leaf access sequence as a function of the number of levels cached for TINY-OAT and Path ORAM, respectively. The error bars represent the lower and upper quartiles centered about the median. These results confirm the analytical results from Section 4.2. In each sub-figure, we consistently see




(a) S3 (ping=40ms) (b) SFTP (ping=65ms)

Figure 4.6: Access time for S3 and SFTP implementations.

TINY-OAT outperform Path ORAM in terms of timing by a factor of at least 2 for all settings of tree-top caching, and with speedup being closer to a factor of 4 or more once half of the levels are tree-top cached. In addition, we see that the SFTP case performs at least an order of magnitude faster than the S3 case, despite having a slower ping time.

Our implementations are highly optimized to take advantage of I/O bound parallelism that is made possible through tree-top caching. The details of this can be seen in the PyORAM source code. Significant profiling shows that the amount of time spent performing encryption and other operations on the client is negligible compared to the time spent waiting to read and write packets. This case study clearly shows that in the cloud setting there is no need to move to lower level languages such as C, C++, or Java when implementing ORAM applications as the bottleneck in speed is entirely due to network latency times.

# Chapter 5: Conclusion

In this work, we describe two new ORAM-based schemes referred to as OAT and TINY-OAT, which achieve highly efficient oblivious accesses to trees by publicizing that the client program accesses a tree data structure. Our schemes introduce various innovative strategies such as level-based ORAM accesses to nodes and heap unification techniques optimized for caching strategies. Without tree-top caching, our schemes are as efficient as Path ORAM with the top half of the storage heap cached (i.e., a factor of 2 reduction in transmission cost). With tree-top caching, our schemes achieve a 7.4 times reduction in online transmission cost over the base case with Path ORAM. This makes OAT and TINY-OAT the most practical oblivious access to tree mechanisms known to date. Such specialized ORAM schemes are ideal to support private access to tree-based search indexes, which may find various applications such as in encrypted database access and searchable encryption.

In addition, we have released a new open-source Python software package named PyORAM that will enable future researchers to easily test and improve upon current ORAM schemes tailored for the cloud storage setting. We believe this will serve as a useful tool for future students and researchers, and help foster new innovations in ORAM for cloud storage.

# Bibliography

[1] Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 181–190. ACM, 2010.

[2] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *Public-Key Cryptography–PKC 2014*, pages 131–148. Springer, 2014.

[3] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.

[4] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. 2011.

[5] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography*, pages 175–204. Springer, 2016.

[6] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 668–679. ACM, 2015.

[7] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawcyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21th Annual Network and Distributed System Security Symposium — NDSS 2014*. The Internet Society, February 23-26, 2014.

[8] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*, pages 353–373. Springer, 2013.

[9] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *Communications (ICC), 2012 IEEE International Conference on*, pages 917–922. IEEE, 2012.

[10] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442–455. Springer, 2005.

[11] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Advances in Cryptology–ASIACRYPT 2014*, pages 62–81. Springer, 2014.

[12] Kai-Min Chung and Rafael Pass. A simple ORAM. Technical report, DTIC Document, 2013.

[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 79–88. ACM, 2006.

[14] Jonathan Dautrich and Chinya Ravishankar. Combining ORAM with PIR to minimize bandwidth costs. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 289–296. ACM, 2015.

[15] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, 2014.

[16] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography*, pages 145–174. Springer, 2016.

[17] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, and Srinivas Devadas. Tiny ORAM: A low-latency, low-area hardware ORAM controller. Technical report, Cryptology ePrint Archive, Report 2014/431, 2014. http://eprint. iacr. org.

[18] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, and Srinivas Devadas. RAW path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[19] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2013.

[20] Eu-Jin Goh et al. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[21] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[22] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[23] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming*, pages 576–587. Springer, 2011.

[24] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

[25] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious storage with low I/O overhead. *arXiv preprint arXiv:1110.1851*, 2011.

[26] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 13–24. ACM, 2012.

[27] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 157–167. SIAM, 2012.

[28] Gabriel A. Hackebeil. A case study on oblivious access to tree data structure. https://github.com/ghackebeil/OTAExperiments, 2016.

[29] Gabriel A. Hackebeil. PyORAM. https://github.com/ghackebeil/PyORAM, 2016.

[30] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[31] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial cryptography and data security*, pages 258–274. Springer, 2013.

[32] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[33] Jacob R Lorch, Bryan Parno, James W Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, volume 2013, pages 199–213, 2013.

[34] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[35] Muhammad Naveed. The fallacy of composition of oblivious RAM and searchable encryption. Technical report, Cryptology ePrint Archive, Report 2015/668, 2015.

[36] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic searchable encryption via blind storage. In *35th IEEE Symposium on Security and Privacy*, pages 48–62, May 2014.

[37] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523. ACM, 1990.

[38] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology–CRYPTO 2010*, pages 502–519. Springer, 2010.

[39] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *Usenix Security*, 2015.

[40] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 571–582. ACM, 2013.

[41] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.

[42] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[43] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.

[44] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652*, 2011.

[45] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[46] Yinqi Tang, Dawu Gu, Ning Ding, and Haining Lu. Phrase search over encrypted data with symmetric encryption scheme. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 471–480. IEEE, 2012.

[47] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[48] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202. ACM, 2014.

[49] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.

[50] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148. ACM, 2008.

[51] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 977–988. ACM, 2012.

[52] Attila A. Yavuz and Jorge Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *Selected Areas in Cryptography (SAC 2015)*, Lecture Notes in Computer Science. Springer International Publishing, August 2015.