NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM

THESIS TITLE: Practical Obfuscating RAM and its Applications

AUTHOR: Travis Maybury

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor Date: 08/08/15

Thesis Reader: 06/24/15

Thesis Reader: 06/26/15

Thesis Reader: 06/26/15

Date: 7/01/15

GRADUATE SCHOOL APPROVAL:

Director, Graduate School: 08/07/15

Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature: 8/7/15

Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Practical Oblivious RAM and its Applications

by

Travis Mayberry

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Department of Computer Science

August 2015
Abstract

Department of Computer Science

Doctor of Philosophy

by Travis Mayberry
Motivated by various reasons, including cost advantages, reliability and convenience, individuals and corporations are frequently outsourcing their data to public cloud platforms such as Amazon AWS, Apple iCloud, and Microsoft Skydrive. If a user wishes to ensure privacy of their data from the cloud provider, they must of course encrypt it before uploading. Unfortunately, this might not be enough. When, where and how the data is accessed from the cloud can often reveal as much or more private information than the data itself. This access pattern also needs to be hidden from the cloud provider to ensure maximum privacy.

Currently, the main cryptographic construction used to provide this hiding is called Oblivious RAM. When the user accesses data on the server, they also shuffle and reencrypt portions of the data so that two accesses to the same file can’t be recognized as being the same. Although ORAM constructions have existed with good asymptotic complexity, there remain several barriers to adoption in regards to practical performance and usability. This dissertation addresses some of those problems in an effort to make ORAM practical for real world applications.

First, we study how bandwidth can be drastically reduced by taking advantage of not only the storage ability of the cloud, but its computational capabilities as well. Using recent advances in homomorphic encryption, ORAM schemes can be augmented to trade communication complexity for server computation, which is comparatively very cheap on current cloud platforms.

Beyond the traditional concern of communication complexity, there are additional usability problems that ORAM constructions have to consider. We show how constructions relying initially on a fixed size database (an unfortunately requirement for the cloud, which counts scalability as one of its main assets) can be expanded to allow dynamic resizing. We also show how to create the first ORAM construction which is secure for multiple concurrent users. Previous schemes support only a single user.

Finally, we show that for some specific use cases a restricted type of write-only ORAM can be used to achieve sufficient privacy while drastically reducing the user’s overhead. We also show that this ORAM is independently interesting for hidden volume disk encryption, and provide some of the first formal definitions for such encryption schemes.
Acknowledgements

I would like to especially thank my advisors Agnes Chan and Erik-Oliver Blass for going above and beyond for me. I can’t imagine going through the last few years without them.

I would also like to thank and acknowledge Tarik Moataz as my co-author and collaborator for the work on Resizable ORAM in chapter 4, and Kaan Onarlioglu as a co-author and programmer for the implementation of our work in chapter 5. Additionally, Guevara Noubir as co-author for the work in chapters 4-6. I have been lucky enough to work with some amazing colleagues at Northeastern University and I am grateful beyond measure to all of them.
Contents

Abstract

Acknowledgements

1 Introduction

1.1 Background

1.1.1 Square-root ORAM

1.1.2 Hierarchical Construction

1.1.3 Tree-based ORAM

1.1.4 Path ORAM

2 Private Information Retrieval for the Cloud

2.1 Problem Statement

2.2 Motivation

Overview

Comparison of existing work

Our approach

2.3 PIRMAP

2.3.1 PIRMAP Specification

2.3.2 Encryption Scheme

2.3.3 Privacy Analysis

Security of Trostle-Parrish PIR

2.4 Evaluation

2.4.1 Setup

Local

Laptop

Amazon

2.4.2 Results

2.5 Conclusion

3 Combining PIR and Oblivious RAM

3.0.1 Motivation

3.1 Path-PIR’s Hybrid Construction

3.1.1 Oblivious Outsourced Storage

3.1.2 Private Information Retrieval

3.1.3 PIR-Writing
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Opportunistic Hiding Security</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>Write-only ORAM</td>
<td>74</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Basic Write-Only ORAM Construction</td>
<td>74</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Stash-Optimized Write-Only ORAM</td>
<td>77</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Recursive map</td>
<td>78</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Write Complexity</td>
<td>78</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Security Analysis</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>Practical Hidden Volume Encryption with HIVE</td>
<td>80</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Uniform vs. Non-uniform Blocks</td>
<td>80</td>
</tr>
<tr>
<td>5.4.2</td>
<td>HIVE: Combining Volumes</td>
<td>81</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Discussion</td>
<td>83</td>
</tr>
<tr>
<td>5.5</td>
<td>Implementation</td>
<td>84</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusion</td>
<td>86</td>
</tr>
<tr>
<td>6</td>
<td>Multi-client Oblivious RAM</td>
<td>88</td>
</tr>
<tr>
<td>6.1</td>
<td>Multi-client ORAM</td>
<td>89</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Security Definition</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Multi-client Security for</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>Classical ORAMs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Challenge</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>Rationale</td>
<td>93</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Details</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>Note on complexity</td>
<td>96</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Security Analysis</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>Deamortizing</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>Multi-client security</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Tree-based Construction</td>
<td>98</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Overview</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>Challenge</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>Rationale</td>
<td>99</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Details</td>
<td>100</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Security Analysis</td>
<td>103</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Complexity</td>
<td>104</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Conclusion</td>
<td>105</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Cloud computing and cloud storage are becoming an attractive option for businesses and governmental organizations in need of scalable and reliable infrastructures. Cloud providers, e.g., Amazon or Google, have substantial expertise and resources, allowing them to rent their services at very competitive prices. Cloud users are drawn by the ability to pay for only what they need, but maintain the ability to scale up if requirements change. Users can now take advantage of reliable storage solutions without investing large amounts of money for data centers upfront.

Unfortunately, there is a significant downside to storing data in the cloud. Cloud providers cannot always be fully trusted and may not treat sensitive user data very carefully. Seeing news of high-profile hacking incidents involving data theft has become commonplace [1, 2]. Encryption of data at rest provides a partial solution to this problem, but it is not sufficient. Even if the cloud cannot read the encrypted data, it may be able to learn valuable information based on when and how often users access their data. We call this information the user’s “access pattern”. As a motivating example, consider a hospital that outsources patient records to the cloud in order to save on replication and IT costs. If the cloud server sees that, e.g., an oncologist accesses a patient’s data, they can learn with some degree of certainty that this patient has cancer. An adversary could aggregate information on data accesses to learn potentially important secrets. As it is generally difficult to quantify what external knowledge adversaries may have and what inferences they could make, it is important to hide a user’s access pattern as well as the data being accessed.

There are traditionally two ways to hide a user’s access pattern: Oblivious RAM (ORAM) [3] and Private Information Retrieval (PIR) [4]. The traditional approach taken by ORAM is to arrange the data in such a way that the user never touches the same piece twice, without an intermediate “shuffle” which erases the correlation between locations in the storage. ORAMs have historically featured low amortized communication complexity and did not require any computation on the server, but occasionally the user was required to download and reshuflle the
entire database. This could become impractical in cloud scenarios, especially if the user is a low-powered or communication-constrained device.

Private Information Retrieval, in contrast with ORAM, does not rely on shuffling of the data but instead uses homomorphic encryption to privately select out the data requested by the user. The user generates encrypted requests and sends them to the server. Since PIR does not try to hide a sequence of accesses, but each access individually, the amortized cost is equal to the worst-case cost. Unfortunately, the requirement that the server computes over the entire database for each query is often impractical, especially for large databases.

Fortunately, there has been a recent flurry of research on Oblivious RAM, achieving many improvements that were long thought difficult or impossible. Chief among them being sublinear worst-case complexity guarantees, accomplished by Shi et al. \cite{shi2014} and independently by Kushilevitz et al. \cite{kushilevitz2014}, but with very different approaches. Since then, several additional schemes have been proposed that achieve better communication complexity, but at the cost of increasing client memory from constant to logarithmic \cite{stefanov2014} or polynomial \cite{pathoram2014, pathoram2015} in \(n\), the number of blocks in the database.

The works of Shi et al. \cite{shi2014} and Stefanov et al. \cite{stefanov2014} are especially interesting because they use an entirely new paradigm for Oblivious RAM: a tree-based construction in which data blocks are inserted at the root and incrementally filtered down to the leaf nodes, making room for future operations at the top of the tree. This has the benefit of giving their scheme a worst-case complexity that is equal to the average case, which is a significant gain in terms of practicality. It also allows for large gains in performance. However, the current leading ORAM construction, Path ORAM\cite{pathoram2014}, might still be considered too costly to use in a real-world situation. For example, on common databases it may impose up to a 200x overhead on communication. Since cloud deployment is currently used to save money on infrastructure costs, any potential user must be very wary of bandwidth and computational overhead which could easily outweigh any cost savings.

Moreover, the change from amortized to worst-case complexity has added an additional problem that impedes its widespread use: it is no longer easy to resize ORAMs. Previously, one could simply choose a new size when the database was being shuffled. Now, however, it would ruin the worst case complexity of the schemes if one has to process the entire database at once.

In this thesis, we develop more practical solutions for privacy-preserving data outsourcing, which include lower overhead and improved costs when executed on cloud platforms. Beyond simple query overhead, there are also other important matters to consider before a system is ready for practical deployment. Common cloud architectures are massively parallel and distributed, meaning that a successful protocol should be easily parallelizable and adaptable to distributed computing platforms, e.g. MapReduce.
We start in chapter 2 with a Private Information Retrieval protocol that is specifically designed for cloud deployment, including an implementation which uses MapReduce to parallelize execution over distributed systems. Analysis shows that this scheme (and PIR in general) is particularly suited for distributed computing settings, and allows for private retrieval in less time than an existing naive approach which simply downloads the entire database. Additionally, we also show that it is actually cost-effective when used on existing cloud infrastructure.

This result leads into chapter 3 where we use PIR as a subroutine to create a more efficient Oblivious RAM. Taking advantage of computational ability on the cloud allows for a dramatic decrease in bandwidth overall, and a particularly low online complexity which is the amount of time the user perceives a query to take. We again show that this scheme can be deployed in a cost-effective manner on Amazon AWS, and that it beats comparable schemes in bandwidth, query time and monetary cost.

In chapter 4 we address the problem of resizing modern ORAMs. We show that secure online resizing can be done for tree-based ORAMs, and that the optimal algorithm depends on how often and by how much the user wishes to resize.

Chapter 5 considers the setting where the ORAM adversary does not see all read and write operations done to the storage media, but only the writes. Several important use cases, including hard drive encryption and cloud backup, can make good use of this relaxed setting. Furthermore, under this restriction, we show that it is possible to achieve optimal $O(1)$ communication complexity with very small constants, making ORAM truly practical for any database size. To further motivate the importance of write-only ORAM, we examine the problem of hidden volume encryption, where a user wishes to not only encrypt their data, but be able to plausibly deny that the data exists in the first place. We develop the first formal security definition for this setting and show that a construction based on write-only ORAM can be proven secure while existing techniques cannot.

Finally, chapter 6 introduces the first construction of a multi-client Oblivious RAM that is secure against malicious adversaries. ORAMs have not traditionally be able to support multiple clients because they are reliant on shuffling of data between accesses. A malicious server can choose to not propagate these changes between clients and force them to perform operations which reveal information about their shared access pattern. If the clients do not have some out-of-band method of communication, there is little they can do against this type of attack using traditional ORAMs. We show that, with insight gained from our write-only ORAM construction, it is possible to create a secure multi-client ORAM by making a client’s operations not only independent of their access pattern, but independent of the state of the data on the server. This means that even if the adversary changes something and presents a false view to the client, it cannot impact the moves that the algorithm makes and so the malicious adversary does not gain an advantage.
With this work, we have addressed several important barriers to adoption for Oblivious RAM and pushed the state-of-the-art closer to a state which is truly practical and usable. However, our optimally efficient solution only works under a write-only security model, and although we have reduced overhead in the general case it still remains a non-trivial drawback of ORAM. Additionally, although we present a new construction for ORAM which achieves multi-user security for the first time, it is significantly less efficient that the state-of-the-art single user schemes and leaves the door open for further improvement.

1.1 Background

We start by reviewing the concept of Oblivious RAM and some of the relevant existing work on the topic. An Oblivious RAM algorithm provides a client access to a block-based data store which holds \( N \) blocks of \( B \)-bit data. The client can perform two operations: \( \text{Read}(x) \), to read from the block at address \( x \) and \( \text{Write}(x, v) \) to write the value \( v \) to block \( x \).

The goal of an Oblivious RAM is to provide an interface to the \( N \) blocks such that the untrusted device which is storing the data (i.e. the disk drive, cloud provider, etc.) does not learn anything about the client’s pattern of accesses to that data. Each access is translated by the ORAM algorithm into one or more accesses on the storage device which somehow hide the overall pattern of accesses that the client is executing. Formally, we start by defining a single client access, or operation.

**Definition 1.1** (ORAM Operation \( \text{OP} \)). An operation \( \text{OP} \) is defined as \( \text{OP} = (o, x, v) \), where \( o = \{ \text{Read}, \text{Write} \}, x \) is the virtual address of the block to be accessed and \( v \) is the value to write to that block. \( v = \perp \) when \( o = \text{Read} \).

Now, we can define an ORAM in terms of its constituent algorithms.

**Definition 1.2** (Secure ORAM scheme \( \Pi \)). A multi-client ORAM \( \Pi \) is defined by tuple \( \Pi = (\text{Init}, \text{Access}) \).

1. \( \text{Init}(\lambda, N, B) \) initializes \( \Pi \). It takes as input security parameter \( \lambda \), total number of blocks \( N \), block size \( B \). \( \text{Init} \) outputs an initial ORAM state \( \Sigma_{\text{init}} \), which encompasses the entirety of the ORAM that is stored on the untrusted storage, and a client state \( st \) which is kept local to the client.

2. \( \text{Access}(\text{OP}, \Sigma, st) \) performs operation \( \text{OP} \) on ORAM state \( \Sigma \). \( \text{Access} \) outputs (1) an access pattern \( \langle (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) \rangle \), where \( (\alpha_j, \nu_j) \) denotes that the block at position \( \alpha_j \) in state \( \Sigma \) is read from or replaced by string \( \nu_j \), and (2) a new state \( st \) for the client.
Finally, we formally define the security of an ORAM scheme against untrusted, potentially malicious storage devices. Consider the following game between an adversary \( A \) and challenger \( C \).

Another popular, more formal way to illustrate this definition is by considering the adversary \( A \) to be a function, which chooses accesses to be run and at the end outputs the bit \( b' \).

**Definition 1.3** (ORAM Security Experiment).

\[
b \overset{\$}{\leftarrow} \{0, 1\} \\
(\Sigma_{\text{init}}, st) \leftarrow \text{Init}(\lambda, n, B) \\
(\Sigma, \text{OP}_0, \text{OP}_1, i, st) \leftarrow A(\lambda, n, B, \Sigma_{\text{init}}) \\
\text{for } j = 1 \text{ to } poly(\lambda) \text{ do} \\
\quad (st, <(\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m)>) \leftarrow \text{Access}(\text{OP}_b, \Sigma, st) \\
\quad (\Sigma, \text{OP}_0, \text{OP}_1, i, st_A) \leftarrow A(<(\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m)>, st_A) \\
\text{end} \\
b' \leftarrow A(st_A) \\
\text{output } 1 \text{ iff } b = b'
\]

In both cases, we say that an ORAM \( \Pi = (\text{Init}, \text{Access}) \) is secure iff for all PPT adversaries \( A \)

\[
Pr[A \text{ outputs } b' = b] < \frac{1}{2} + \epsilon(\lambda),
\]

where \( \epsilon \) is a negligible function in security parameter \( \lambda \).
In straightforward terms, this definition means that any two sets of accesses which are the same length will be indistinguishable to the untrusted storage device (adversary) when executed with a secure Oblivious RAM. This notion is captured by allowing the adversary to adaptively choose two sets of accesses. The worst-case scenario is when the adversary has complete knowledge of the client’s accesses, having chosen the possibilities himself, except for a single bit of information which indicates which of the two sets are actually being executed. An ORAM is secure if, given even this advantage, the adversary cannot determine that last bit of information with greater than 50% probability.

One can also consider an ORAM which is secure not against malicious adversaries, but semi-honest or honest-but-curious adversaries. These are adversaries which will try to learn something about the client’s access pattern but will otherwise behave according to the protocol. In the above definition, the adversary does not have to follow the protocol and can in fact modify the ORAM state, ∆, in any way that they choose. A definition for semi-honest adversaries would stipulate that ∆ must not be modified by the adversary.

We now move on to discussion of existing ORAM solutions and how the state-of-the-art has progressed over time.

1.1.1 Square-root ORAM

The concept of an Oblivious RAM, along with its first construction, was presented by Goldreich [10]. In this ORAM, storage is divided into two partitions. One holds \( N + \sqrt{N} \) blocks, and we will call this main memory. The other partition holds \( \sqrt{N} \) blocks, the cache. When the ORAM is initialized, a random permutation \( \pi \) on \( N \) elements is sampled, and each block \( x \) is stored encrypted in the main memory at location \( \pi(x) \). In addition, we store \( \sqrt{N} \) encrypted dummy elements at positions \( \pi(N + 1), \ldots, \pi(N + \sqrt{N}) \). The cache is also encrypted, initially holding \( \sqrt{N} \) empty blocks. Every access Read(\( x \)) that the (single) client \( u \) makes is then broken down into two steps. First, \( u \) retrieves the cache in its entirety, decrypts it, and searches to see whether the cache contains block \( x \). If it is not, \( u \) retrieves block \( \pi(x) \) from the main memory and inserts it into the local cache. If \( x \) is already in the cache, \( u \) simply reads its value from the local cache and performs an additional read of the next dummy element \( \pi(x'), x' \in \{N + 1, \ldots, N + \sqrt{N}\} \), from main memory. This hides the fact that \( u \) found the block they wanted in the cache. The cache is then finally re-encrypted and stored back in the server.

A Write(\( x, v \)) to the ORAM is implemented similarly. client \( u \) again downloads the cache, decrypts it, and stores block \( x \) with its new value \( v \) in it. Before re-encrypting and uploading the cache back to the server, \( u \) also reads the next dummy element from main memory. To always know which dummy element is the next dummy element, the client can store an (encrypted) dummy counter inside the cache. To ease exposition, we will omit this detail in our protocol.
description below. For integrity verification, recall that $u$ typically needs to compute and store an authentication token (e.g., a MAC) of the latest version of the cache in their local memory [11].

After $\sqrt{N}$ operations, the cache will become full. At that point, the client downloads the cache and the entirety of the main memory. The client merges main memory and cache together by emptying the cache and reshuffling main memory with a new permutation $\pi$. For blocks which are in both the cache and main memory, the cache version will be newer and so is taken to replace the old version.

For blocks of size $B$ bit, the amortized complexity of this scheme is $\tilde{O}(\sqrt{N} \cdot B)$, and worst-case complexity is $O(N \cdot B)$. Each access requires $O(\sqrt{N})$ blocks, but every $O(\sqrt{N})$ operations require an $O(N)$ access. Security is straightforward, because the client never reads the same block twice from main memory without an intervening complete shuffle. The cache is read and re-encrypted in its entirety for each operation. In effect, the cache acts like a “trivial” ORAM, where the entire data store is retrieved and reencrypted on each access.

### 1.1.2 Hierarchical Construction

In addition to the square-root ORAM, Goldreich [10] also propose a generalization which achieves poly-log overhead. In order to do this, it has a hierarchical series of caches instead of a single cache. Each cache has $2^j$ slots in it, for $j$ from 1 to $\log N$, where each slot is a bucket holding $O(\log N)$ blocks. At the bottom of the hierarchy is the main memory which has $2 \cdot N$ blocks.

The main idea is that each level of the cache is structured as a hash table. Up to $2^{j-1}$ blocks can be stored in cache level $j$, half the space is reserved for dummies like in the previous construction. After $2^{j-1}$ blocks, the entire level is retrieved and shuffled into the next level. Shuffling involves generating a new hash function and rehashing all the blocks into their new locations in level $j + 1$, until the shuffling percolates all the way to the bottom and the client must shuffle main memory to start again. Level $j$ must be shuffled after $2^{j-1}$ accesses, resulting in an amortized poly-logarithmic cost.

To access a block, a client queries the caches in order using the unique hash function at each level. When the block is found, the remainder of the queries will be on dummy blocks to hide that it was found. After reading, and potentially changing the value of the block, it is added back into the first level of the cache and the cache is shuffled as necessary.
1.1.3 Tree-based ORAM

There exists a large body of work on improving Oblivious RAM since the original concept introduced by Goldreich [10]. For example, Pinkas and Reinman [12] and Boneh et al. [13] have reduced amortized communication to poly-logarithmic complexity. However, even if these constructions feature low amortized cost, worst-case complexity was still $O(N \cdot B)$, which is prohibitive in many scenarios. This is due to the fact that, after a certain number of operations, the entire database needs to be downloaded and reshuffled by the user. One of the drawbacks of the original Oblivious RAM algorithm and its derivatives is that the overhead guarantees are only amortized and not worst-case. This means that, occasionally, the client will have to do a lot more work than they expect during an access in order to maintain the state of the ORAM. In fact, after a number of operations they have to download and reshuffle the entire database, which is very impractical for real-world use.

Fortunately, Shi et al. [5] introduced a new type of Oblivious RAM which is able to give worst-case guarantees as well as amortized. This is accomplished by storing the data in tree form and doing a partial shuffling of the tree after every access.

To start, instead of supporting the traditional two operations of ORAM, Read and Write, tree-based ORAMs have a slightly different interface:

1. **ReadAndRemove**(x) – Returns the value of the block with identifier x, or a unique identifier ⊥ if x identifies a dummy or if x does not exist in the ORAM. Additionally, this operation removes block x from the ORAM.

2. **Add**(x, y) – Adds a block with identifier x and value y to the ORAM.

Using this interface, one can emulate the traditional one. A Read can be implemented by calling **ReadAndRemove** followed by **Add** to put the block back in the ORAM. Similarly, **Write** can be emulated with a **ReadAndRemove** (on a dummy value, if the block does not exist in the ORAM, yet) and an **Add** with the new value of the block. Conceptually, this set of operations is more conducive to an ORAM construction, because it hints at the idea that, when reading a block, there must be an active relocation of that block in order to disassociate future accesses to it.

The ORAM of Shi et al. [5] also has two other operations:

1. **Pop**( ) – Returns a real data block if the ORAM contains such a block and a dummy otherwise.

2. **Evict**( ) – Partially shuffles the ORAM in order to maintain a consistent and sustainable state. This must be executed after ever access.
Assume for simplicity that \( N \) is a power of two. In order to amortize the cost of shuffling, Shi et al. [5] use a tree of \( 2N - 1 \) “bucket” ORAMs arranged in a tree of depth \( \log N \). These internal ORAMs are each fully-functioning ORAMs with a capacity of \( n := \log N \) “slots”. The buckets must have three properties: (1) support a non-contiguous identifier space (2) support \( \text{ReadAndRemove} \) and \( \text{Add} \), and (3) have zero probability of failure. In Path-PIR, we will replace the bucket ORAMs with PIR operations, so these are the three properties our construction must have in order to be sound.

When blocks are added to the ORAM, they are inserted in the root bucket. Each block is tagged with a random number \( t \in \{0, \ldots, N - 1\} \), which corresponds to a leaf node towards which that block will be moving. The user stores a map \( M \) which, for each block in the ORAM, contains the value \( t \) for that block. \( M(x) \) denotes the value \( t \) for \( x \), stored in the user memory. As this would imply \( O(N) \) user memory, Shi et al. [5] show how this map can itself be recursively stored in an ORAM to achieve \( O(1) \) client memory. However, for the sake of clarity, we will assume \( O(N) \) user memory when presenting Path-PIR. The recursive technique can be applied equally to our construction, since it does not depend on the makeup of the individual buckets. This adds a \( \log N \) factor to each query, because there are at most \( \log N \) recursive ORAMs to store that map.

\( \text{ReadAndRemove} \) – Assuming that a block \( x \) starts at the root bucket and moves down the tree towards its respective leaf node, block \( x \) will always be found somewhere along the path from the root to \( M(x) \), denoted \( \mathcal{P}(M(x)) \). Therefore, a \( \text{ReadAndRemove} \) can be performed by executing \( \text{ReadAndRemove}(x) \) on every bucket along the path from the root to \( M(x) \). One bucket will store block \( x \). Block \( x \) will be removed from this bucket, and all other buckets along the path will return \( \bot \).

\( \text{Add} \) – A new leaf node \( t \leftarrow \{0, \ldots, N - 1\} \) is randomly chosen, and the user inserts block \( x \) with value \( y \) into the root bucket, tagged with leaf node \( t \).

Every Read and Write operation consists of one \( \text{ReadAndRemove} \) and one \( \text{Add} \). Two Read or Write operations to the same block will be completely independent, because a new random \( t \) is chosen for each \( \text{Add} \). Therefore, this construction achieves obliviousness.

**Tree balancing.** To facilitate the movement of blocks towards leaf nodes, and to prevent internal buckets from overflowing, the user must Evict blocks from internal buckets to their children. At each level of the tree, the user randomly picks \( \nu \in \mathbb{N} \) buckets and executes \( \text{Pop} \) to read and remove one data block from them. The user then writes to each of the child buckets, moving data blocks toward the correct leaf nodes and performing dummy operations on those children which are not on the correct path maintaining obliviousness. One can show that \( \nu = 2 \) is sufficient to keep any buckets from overflowing with high probability, if Evict is performed after every Read or Write operation [5].
**Complexity.** Assuming each bucket ORAM with individual capacity of $n = \Theta(\log N)$ has communication complexity $R(n)$ for its operations, we can calculate the overall cost for this tree construction. ReadAndRemove performs one operation on each of the $\log N$ buckets, so its cost is $\log N \cdot R(n)$. Add operates only on the root bucket, and so has complexity simply $R(n)$. Evict operates on $3 \cdot \nu$ buckets (one parent and 2 children for every bucket evicted) on each level of the tree and so has cost $3 \cdot \nu \cdot \log N \cdot R(n)$. For all bucket ORAMs, the worst-case cost is $O(n)$. For the individual buckets, $n = \log N$, so the worst-case cost for eviction (the most expensive operation) is $3 \cdot \nu \cdot \log N \cdot \log N$. Therefore, regardless of which bucket construction is used the overall worst-case complexity is $O(B \cdot \log^3 N)$. Recursively storing the user memory requires at most $\log N$ additional ORAMs, adding another $\log N$ factor to the overall cost resulting in $O(B \cdot \log^3 N)$.

### 1.1.4 Path ORAM

Although the previous tree-based construction has, for the first time, worst-case polylogarithmic guarantees, it still requires a substantial amount of overhead, $O(\log^3 N)$. Additionally, the storage overhead (multiplicative factor governing the amount of storage needed) is large at $O(\log N)$. Fortunately, these problems were later addressed by Stefanov et al. [7] in their construction, Path ORAM.

As in the previous construction, the RAM in Path ORAM is structured as a tree with $N$ leaf nodes. The difference in Path ORAM is that each node in the tree holds up to $Z$ blocks where $Z$ is a small constant, instead of $\Theta(\log N)$. This makes the storage overhead only a constant. Each block in the ORAM is tagged with a value uniform in the range $[0, N)$. As an invariant, blocks will always be located on the path from the root of the tree to the leaf node corresponding to their tag. Over the lifecycle of the tree, blocks will enter at the root and filter their way down toward the leaves, making room for new blocks to in turn enter at the root. As before, the client has a map which stores, for every block, which leaf node it is tagged for.

**ReadAndRemove:** To retrieve block $x$, the client looks up in the map which leaf node it is tagged for and retrieves all nodes from the root to that leaf node, denoted $P(M(x))$. By the tree invariant, block $x$ will be found somewhere on the path $P(M(x))$. The client then removes block $x$ from the node it was found in, reencrypts all the nodes and puts them back in the RAM.

**Add:** To put a block back in the ORAM, the client simply retrieves the root node and inserts the block into one of its free slots, reencrypting and writing the node back afterwards. The map is updated with a new random tag for this block in the interval $[0, N)$. With the small bucket size, there is a non-trivial chance that the root bucket may be full. If there is not enough room in the root node, the client keeps the block locally in a “stash”, waiting for a later opportunity to insert it into the tree.
Evict: So that the stash does not become too large, after every operation the client also performs an eviction which moves blocks down the tree to free up space. Instead of evicting independent buckets at each level like the previous construction, eviction consists of picking a path in the tree (using reverse lexicographic ordering [14]) and moving all blocks on that path as far down the tree as they can go, without violating the invariant. Additionally, the client inserts any matching block from the stash into the path.

As an additional contribution, Stefanov et al. [7] show that if $B = \Omega(\log^2 N)$, the map can be recursively stored in a careful way to eliminate its addition to the overall complexity. This results in a total overhead of $O(\log N)$, compared to $O(\log^3 N)$ previously. However, in relation to the $O(\log^3 N)$ scheme, Path ORAM requires non-constant client memory. In the previous construction, the client only needed to store locally a single block which begins the recursive indexing into the map. In Path ORAM, the client has a stash of blocks stored locally which varies in size. This is an important distinction between the two schemes, and makes the original one still interesting despite the fact that it has larger overhead compared to Path ORAM.

Having covered the relevant existing work in ORAM we now continue to our contributions, starting with work on the related cryptographic primitive Private Information Retrieval.
Chapter 2

Private Information Retrieval for the Cloud

Although the main topic of this thesis is Oblivious RAM, our first original contribution comes from examining a related technique, Private Information Retrieval. In chapter 3 we show that PIR can be combined with ORAM to reduce its overhead, but first we discuss in this chapter how PIR can be done efficiently in a cloud setting.

Private Information Retrieval (PIR) offers a solution to the information leakage problem by hiding access patterns [4, 15], similar to ORAM. PIR stores a database at a server holding $N$ files, or blocks, each of size $B$ bits, and a user can query for one $B$ bit block without leaking which block to the server. The key difference between ORAM and PIR is that in PIR the database is stateless; that is, it is not shuffled like in ORAM. A trivial way to accomplish this is to simply transmit the entire database to the user. However, this is communication inefficient, thus PIR focuses on achieving lower communication bounds. Unlike ORAM, computational cost of every PIR scheme is necessarily $O(N \cdot B)$, as the server must “touch” every piece of the database if the server is to remain oblivious of the requested piece.

In previous iterations, the server’s computation is comprised of expensive cryptographic operations over the entire database. Because of the significant overhead this imposes, it has recently been questioned whether PIR will ever become practical in a real-world cloud computing setting, cf. Sion and Carbunar [16]. Typically, cloud providers such as Amazon charge their customers for both data transfer and CPU hours [17]. Due to the necessary condition that PIR protocols compute over the entire database for each query, it has been argued that trivial PIR (retrieving the whole $(B \cdot N)$ bit database) is not only faster, but also cheaper for the cloud customer compared to a PIR query that involves lengthy computation [16, 18, 19].
Another open question is how to perform PIR in a real-world cloud computing environment. In contrast to a single machine server storing the data, one of the biggest challenges in cloud computing is a design that scales easily to the large distributed systems which are characteristic in cloud settings. In order to alleviate this difficulty, major cloud providers such as Amazon, Google, IBM and Microsoft offer an interface to the prominent MapReduce [20] API for distributed computing to their users. MapReduce comprises not only parallelization (“Map”) of work, but an aggregation (“Reduce”) of individual results to keep computational burden on the user side low.

This chapter considers the single-server, computationally-private information retrieval setting. This is appropriate, because, although a cloud provider may allow access to many servers, they must all be considered as a single trusted entity: they are under the control of the same organization. Trust-wise, the cloud should be viewed as a single, large server with many distributed CPUs. In the single-server setting, it is known that unconditionally secure PIR cannot be more efficiently achieved than transferring the entire database [15], so we will be concerned instead with a computationally secure PIR protocol. Further use of the term PIR will be in reference to computationally-secure PIR, unless otherwise noted.

We present PIRMAP, an efficient single-server cPIR protocol suited for cloud computation platforms. PIRMAP especially targets retrieval of relatively large files, a more specific setting than considered by previous work[4, 21]. In a scenario with \( N \) files each of size \( B \) bits and \( B \gg N \), PIRMAP achieves communication complexity linear in \( B \) with low constants. PIRMAP is designed for and leverages MapReduce parallelization and aggregation. We have implemented PIRMAP in Hadoop MapReduce, and its performance will be presented in Section 2.4.

Our contributions in this chapter are:

1.) An analysis of existing work with respect to its practicality in a MapReduce cloud setting.

2.) The design of PIRMAP, an efficient, practical PIR scheme using MapReduce that achieves optimal communication complexity \( O(B) \) when retrieving an \( B \) bit file for \( B \gg N \). Additionally, PIRMAP has computational complexity as good as or better than currently known protocols by using a more efficient homomorphic encryption. PIRMAP runs on top of standard MapReduce, not requiring changes to the underlying cloud infrastructure.

3.) An implementation of PIRMAP that is usable in real-world MapReduce clouds today, e.g., Amazon. We evaluate PIRMAP, first, in our own (tiny) local cloud and, second, with Amazon’s cloud. We verify its practicality by scaling up to 1 TByte of data in Amazon’s cloud setting, in comparison to the previously largest single database PIR experiment with up to 28 GByte of data [18]. This demonstrates the efficiency and practicality of PIRMAP in the real-world. PIRMAP is more than one order of magnitude cheaper and faster than trivial PIR, and – in the
case where the file size is much bigger than the number of files – we can significantly outperform other known schemes. PIRMAP’s source code is available for download [22].

2.1 Problem Statement

**PIR:** A PIR protocol is a sequence of interactions between a user and a server storing \( N \) files that results in the user retrieving one targeted file while the server does not learn which file. More formally, the server cannot guess with probability greater than \( 1/N \) which file was queried. Note that this probability does not increase over multiple queries. This effectively hides the user’s access pattern as each query is computationally indistinguishable from the others. The only information leaked is the number of files queried.

A \( PIR_{B}^{N} \) protocol \((Query, Transfer, Recover)\) is a computationally-secure private information retrieval protocol over a database of \( N \) elements, each of bit length \( B \). The *Query* function generates the query and sends it to the server. *Transfer* is run on the server and involves transforming the received query into the results of the query and sending it back to the client. Finally, *Recover* is run by the client to transform the response from the server into the correct plaintext query result.

**MapReduce:** With the trend towards more computers and more cores rather than faster individual processors, it is important that any practical PIR implementation be deployable in a way that can take full advantage of parallel and cluster computing environments. Perhaps the most widely adopted architecture for scaling parallel computation in public clouds today is Google’s MapReduce [20]. Its design allows for a set of computations, or “job”, to be deployed across many nodes in a cloud. Its biggest advantage is that it scales transparently to the programmer. That is, once an implementation is written using MapReduce, it can run on any number of nodes in the data center, from one up to hundreds or thousands, without changes in the code. This is managed by splitting computation into two phases, each of which can be run in parallel on many computing nodes.

The first phase is called the “Map” phase. MapReduce will automatically split the input equally among available nodes in the cloud data center, and each node will then run a function called *map* on their respective pieces (called *InputSplits*). It is important to note that the splitting actually occurs when the data is uploaded into the cloud (in our case when the patient record/files are uploaded) and not when the job is run. This means that each “mapper” node will have local access to its *InputSplit* as soon as computation has started, thus the lengthy copying and distributing period is avoided. The map function runs a user defined computation on each *InputSplit* and “emits” a number of *key-value* pairs that go into the next phase.
The second phase, “Reduce”, takes as input all of the key-value pairs emitted by the mappers and sends them to “reducer” nodes in the data center. Specifically, each reducer node receives a single key, along with the sequence of values output by the mappers which share that key. The reducers then take each set and combine it in some way, outputting a single value for each key.

Despite being widely used, MapReduce is a very specific computational model and not all algorithms can be easily adapted to it. A practical PIR such as PIRMAP has to take its specifics into account – as we will see below.

2.2 Motivation

Despite the large amount of theoretical research, there has not been much investigation into practical PIR. In 2010, Trostle and Parrish [23] experimented with the state of the art and found that it took at least 8 minutes to retrieve a file of size only 3 MB, out of a data set of 3 GB on commodity hardware. Additionally, existing schemes have significant communication overhead for files of that size and larger. For example, Lipmaa [21], enjoying the most efficient asymptotic communication complexity today, requires 30 MB of communication in the above scenario.

In examining related work, we make two nonstandard observations which apply to our MapReduce cloud setting. First, since computation for PIR must necessarily be \(O(N \cdot B)\), computation will most likely be the bottleneck in any protocol. The time to communicate queries and responses will be relatively small in our setting, compared to the time it takes to calculate over the entire database. Therefore, it may be useful to trade some extra communication for smaller computational burden. The second observation relates to MapReduce itself. Since each mapper may run on a different node in the cloud, communication and synchronization within the cloud are very expensive. To take full advantage of the cloud, there should be few, if any, inter-dependencies between stages of computation.

Overview

One of the first cPIR schemes to achieve sublinear communication is that of Kushilevitz and Ostrovsky [4], using an additively homomorphic cipher \(E\) as follows:

1. The server arranges an \(N\) elements database as a \(\sqrt{N} \times \sqrt{N}\) matrix

2. Query\((x)\) – The user generates a vector \(\vec{v}\) of length \(\sqrt{N}\) where \(\vec{v}_x = E(1)\), and \(\vec{v}_i = E(0)\), \(\forall i \neq x\) (\(E\) here denotes encryption)

3. Transfer\((\vec{v})\) – Vector \(\vec{v}\) is sent to the server. The server multiplies \(\vec{v}\) with the database matrix and returns the result as vector \(\vec{v}'\)
4. Recover($\vec{v}$) – $\vec{v}$ now consists of $\sqrt{N}$ encrypted elements, one of which is the element the user is interested in. The user chooses this element and decrypts it, discarding the rest.

The scheme is sound, because the user sends a vector $\vec{v}$ which “zeroes out” all the rows in the matrix besides the row requested. The communication cost is $O(B \cdot \sqrt{N})$ which is still quite high for large values of $N$. Kushilevitz and Ostrovsky [4] also show that this protocol can be repeated recursively to achieve communication less than $O(N^c)$ for any $c > 0$, but at the cost of worse constants and computational complexity. This “square-root solution” computes only once over the database, but has impractically large communication costs.

Lipmaa [21] reduces communication complexity to $O(B \cdot \log N + k \cdot \log^2 N)$, where $k$ is a security parameter. This is accomplished by generalizing the Kushilevitz and Ostrovsky [4] scheme into a family of protocols, parameterized by a dimension $\alpha$. The Kushilevitz and Ostrovsky [4] scheme can be seen as two-dimensional, because the database elements are arranged in a $\sqrt{N} \times \sqrt{N}$ matrix. If $\alpha = \log N$, the database is viewed as a $\log N$ dimensional $2 \times 2 \times \ldots \times 2$ matrix. In this manner, the user may send a sequence of $\log N$ vectors, each only length 2 which “zero out” half of the remaining database elements at each step. It is conceptually similar to specifying a path in a binary tree, and when the leaf node is reached the only element that will remain non-zero is the one corresponding to that node.

The downside of this scheme is that it requires computing over the entire database twice ($\sum_{i=1}^{\log N} \frac{1}{2^i} = 2$). While not a significant problem asymptotically, as mentioned before, cPIR schemes are usually bottlenecked by their computational cost. Increasing this cost by 100% dramatically effects the efficiency of the scheme, especially given the cloud setting where twice the computation corresponds to twice the monetary cost. Another problem is that the computational complexity of Lipmaa [21] is $O(N \cdot B \cdot k^2)$, because it requires modular exponentiations on ciphertexts of size $k$. This is a significant overhead that we can avoid by using a more efficient homomorphism.

Although Lipmaa [21] requires only one round of communication with the user, it is iterative in that it requires $\log N$ rounds of computation on the server, each round depending on the output of the previous round and operating on a smaller set of data. As we will see, a large part of the cost for MapReduce is in initializing the parallel computation, so restarting MapReduce $\log N$ times adds a significant overhead.

More recently, Aguilar-Melchor and Gaborit [24] have proposed a scheme using lattice-based homomorphic encryption (similar to NTRU[25]) which has much better computational complexity, both asymptotically and in practice. Their scheme also takes advantage of the fact that modern GPUs can solve large parallel problems very quickly. Aguilar-Melchor and Gaborit [24]
are able to achieve fast query response times, but at the cost of a large communication cost. Additionally, their protocol is tuned specifically to the requirements of GPUs and would not scale well in a real-world distributed cloud environment like MapReduce.

Using a different encryption scheme based on the Trapdoor Group Assumption, Trostle and Parrish [23] propose a more traditional PIR scheme based on Kushilevitz and Ostrovsky [4]. They also achieve much faster query response, but again, with large communication costs.

**Comparison of existing work** To understand how existing work compares and is suited to the application targeted in this chapter, we start by examining asymptotic computation and communication complexities in Table 2.1. We can see that Lipmaa [21] has very good communication complexity with relatively bad computational complexity, while Aguilar-Melchor and Gaborit [24] have the opposite. While our protocol PIRMAP has the same asymptotic complexities as that of Aguilar-Melchor and Gaborit [24], this does not adequately describe the costs involved with PIR: we are primarily addressing a concrete, practical setting where constants become very important.

Consequently, in Table 2.2, we present analytically calculated communication costs for specific database and file sizes. Aguilar-Melchor and Gaborit [24] state that their requests are of size $25kb \cdot n$ and the responses are $6 \cdot B$. For databases with $B > N \cdot k$, Lipmaa [21] requires transmitting $\log_2 N$ elements of size $k$ in the request and receiving a response of size $\log N \cdot B$. We chose to use $k = 2048$, as suggested by Lipmaa [21].

In Table 2.2, we can see that even though Lipmaa [21] has good asymptotic communication, there is room for improvement in a concrete setting, especially for databases composed of large files. Aguilar-Melchor and Gaborit [24], similarly to PIRMAP, is designed specifically for these types of databases but it also has relatively high communication costs due to large constants.

**Our approach** In conclusion, we would like to achieve, in real world settings, the small communication cost of Lipmaa [21] with the fast query response times of Aguilar-Melchor and Gaborit [24] and Trostle and Parrish [23]. Lipmaa notes that, when $\alpha = 1$, the PIR scheme is particularly efficient for databases where $B > N \cdot k$. We choose this case as a base for our new protocol PIRMAP, because it requires only one iteration of the database during Transfer and has communication complexity of $O(B + N \cdot k)$. When $B > N \cdot k$, this is optimal, since the server response must always be of size $B$. In short, PIRMAP can be thought of as a combination of this protocol with the fast encryption scheme used by Trostle and Parrish [23]. This new “somewhat” homomorphic encryption scheme requires only modular multiplications rather than exponentiations, significantly improving the asymptotic and real world computational requirements. Unfortunately, after this work was completed it was found that the encryption scheme
TABLE 2.1: Communication and computational complexities of related work.

<table>
<thead>
<tr>
<th></th>
<th>Communication</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kushilevitz and Ostrovsky [4]</td>
<td>$O(\sqrt{n} \cdot B)$</td>
<td>$O(B \cdot N \cdot k^2)$</td>
</tr>
<tr>
<td>Lipmaa [21]</td>
<td>$O(B \cdot \log^2 N)$</td>
<td>$O(B \cdot N \cdot k^2)$</td>
</tr>
<tr>
<td>Aguilar-Melchor and Gaborit [24]</td>
<td>$O(B + N \cdot k)$</td>
<td>$O(B \cdot N \cdot \log k \cdot \log \log k)$</td>
</tr>
<tr>
<td>PIRMAP</td>
<td>$O(B + N \cdot k)$</td>
<td>$O(I \cdot N \cdot \log k \cdot \log \log k)$</td>
</tr>
</tbody>
</table>

TABLE 2.2: Real communication costs of related work in MB for different database sizes.

<table>
<thead>
<tr>
<th></th>
<th>1 MB Files</th>
<th>5 MB Files</th>
<th>10 MB Files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 GB</td>
<td>10 GB</td>
<td>100 GB</td>
</tr>
<tr>
<td>Lipmaa</td>
<td>10</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>Aguilar-Gaborit</td>
<td>29</td>
<td>240</td>
<td>2350</td>
</tr>
<tr>
<td>PIRMAP</td>
<td>3</td>
<td>6</td>
<td>38</td>
</tr>
</tbody>
</table>

by Trostle and Parrish [23] is insecure[26]. This scheme can be replaced by other encryption schemes for similar results, but for simplicity we include here our original work using that scheme. In section 2.3.3 we show results using more secure schemes which indicate that they perform comparably.

2.3 PIRMAP

PIRMAP modifies the PIR protocol by Lipmaa [21], specifically addressing the shortcomings perceived in regards to retrieval of large files in a parallelization-aggregation computation framework such as MapReduce. We start by giving an overview of PIRMAP which can be used with any additively homomorphic encryption scheme.

Upload. In the following, we assume that the user has already uploaded its files into the cloud using the interface provided by the cloud provider.

Query. Our data set holds $N$ files, each of $B$ bits length. Additional parameter $k$ specifies the block size of a cipher. For ease of presentation, we consider the case where all files are the same length, but PIRMAP can easily be extended to accommodate variable length files using padding. PIRMAP is summarized as follows:

1. Query – If the user wishes to retrieve file $x$ out of the $N$ files, he creates a vector $\vec{v} = (v_1, \ldots, v_N)$, where $v_x = E(1)$ and $v_i = E(0) \forall i \neq x$. Here, $E$ is any secure additively homomorphic encryption scheme. The user sends $\vec{v}$ to the cloud.

2. Multiply – The cloud arranges files into a table $T$ as shown in Table 2.3. The cloud divides each file $i$ into $\frac{B}{k}$ blocks $\{B_{i,1}, \ldots, B_{i,\frac{B}{k}}\}$ and multiplies each block by $v_i$, i.e., $B'_{i,j} = v_i \cdot B_{i,j}$. Here, “$\cdot$” denotes scalar multiplication.
TABLE 2.3: Cloud splits files into pieces

<table>
<thead>
<tr>
<th>file</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1</td>
<td>( B_{1,1} )</td>
<td>( B_{1,2} )</td>
<td>...</td>
<td>( B_{1,B} )</td>
</tr>
<tr>
<td>file2</td>
<td>( B_{2,1} )</td>
<td>( B_{2,2} )</td>
<td>...</td>
<td>( B_{2,B} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>fileN</td>
<td>( B_{N,1} )</td>
<td>( B_{N,2} )</td>
<td>...</td>
<td>( B_{N,B} )</td>
</tr>
</tbody>
</table>

3. **Sum** – The cloud adds column-wise to create one result vector \( \vec{r} = (r_1, \ldots, r_B) \). Each element \( r_i \) of that vector is of size \( k \), so the total bit length of \( \vec{r} \) is \( B \) bits. Vector \( \vec{r} \) is an encryption of file \( x \). Vector \( \vec{r} \) is returned to the user who decrypts it.

The cloud performs the matrix-vector multiplication \( \vec{r} = \vec{v} \cdot \mathbf{T} \). In practice, the cloud does not need to create a table, but just needs to perform the block wise scalar multiplications and additions.

The computation consists of multiplying each of the \( N \cdot \frac{B}{k} \) blocks by a value in the request vector and then performing \( (N-1) \cdot \frac{B}{k} \) additions to obtain the final sum. The computational complexity of this scheme is \( O(N \cdot B \cdot M(k) + (N-1) \cdot B \cdot A(k)) \), where \( M(k) \) is the cost of performing one scalar multiplication in the additively homomorphic encryption system, and \( A(k) \) is the cost of one addition. The computational complexity can be broken down into two parts: user to cloud and cloud to user. The user sends a vector of size \( n \) containing ciphertexts of size \( k \), so the bandwidth complexity user-cloud is \( O(N \cdot k) \). The cloud sends back a vector of \( B \cdot \frac{k}{k} \) entries, each of size \( O(k) \), resulting in complexity \( O(B) \). This results in the overall communication complexity of \( O(N \cdot k + B) \).

PIRMAP achieves better overhead than related work [21, 27] in practice, due to the specific values of \( N \) and \( B \) in our setting. Existing work in PIR is efficient only for data sets with large values for \( N \) and small values for \( B \). PIRMAP would do poorly in that setting, because the complexity would be dominated by \( N \). However, PIRMAP’s cloud-to-user communication is optimal at \( O(B) \), because the cloud must send back a message at least the size of the file the user queries for. For values of \( B \) larger than \( N \cdot k \), PIRMAP allows for complexity of \( O(B) \) with small constants. We argue that in practice this is often true. For example, if a user has a 1 TB data set of 10 MB files, \( \frac{B}{N} \approx 800 \). In contrast, under the same conditions, arranging the files in a \( \sqrt{N} \times \sqrt{N} \) matrix would result in a download cost of \( 1024 \cdot 10 \text{ MB} \approx 10 \text{ GB} \). Even if \( N > B \), the actual communication costs are low for practical choices of the parameters, see Section 2.4. Many cloud providers such as Amazon do not charge the user for uploads, but only for downloads. This is an additional benefit of our scheme, because the query result (communication the user is charged for) is always very close to optimal, even when the overall communication is not.

Our assumption of \( B > N \) is realistic and useful for many real-world applications. For example, medical records may contain one or more images (x-ray, MRI, etc) which would make them
several megabytes at the very least. More importantly, as we will see in chapter 3, this scheme can be applied as a subroutine for ORAM algorithms where \( B \) is much larger than \( N \).

**Optimization:** Although the cloud performs most of the computation in this scheme, the user is still required to generate a vector of ciphertexts of length \( n \) and then decrypt the resulting response. As encryption is relatively expensive for additive IND-CPA ciphers, this might require a non-trivial amount of computation that might hurt, e.g., users with low-powered devices such as smartphones. A way to alleviate this problem is to have a moderately powerful trusted server to pre-generate vectors of ciphertexts and upload them to the cloud for the users to use. This trusted server would generate \( m \) “disposable” vectors of size \( n \) such that \( V_{i,j} = \mathcal{E}(1) \) where \( j = \text{PRF}(i) \) and \( V_{i,j} = \mathcal{E}(0) \) otherwise. This allows the user to use one of these disposable vectors at query time and permute it so that the single \( \mathcal{E}(1) \) is at the index of the file it wishes to retrieve. If the key used in the HMAC is shared between the user and trusted server, the user can efficiently locate \( \mathcal{E}(1) \). The user then generates a description of a permutation which moves the \( \mathcal{E}(1) \) value to the correct position and randomly shuffles all other locations. A description of this permutation is of size \( N \cdot \log N \) which is smaller than the size of the vector for \( k > 30 \). This also effectively front loads the upload cost of the query and makes response time even faster.

Although the particular encryption scheme we use (see Section 2.3.2 below) can perform encryptions very quickly and does not require the use of this optimization, we point it out as a general improvement regardless of the cipher used.

### 2.3.1 PIRMAP Specification

In our protocol, the cloud performs two operations: multiplication of each block by the corresponding value in the “PIR vector” \( \vec{v} \), and column-wise addition to construct the encrypted file chosen by the user. These two stages translate exactly to map and reduce implementations respectively. The files will be distributed evenly over all participating nodes where the map function will split each file into blocks and multiply the blocks by the correct encrypted value. The output of these mappers is a set of key-value pairs where the key is the index of the block, and the value is the product of the block and encrypted PIR value. These values are all passed on to the reducers, which take a set of values for each key (block position or column) and add them together to get the final value for each block.

Being interested in \( file_x \), the user executes the above Query to compute \( \vec{v} \) which is sent to the MapReduce cloud. There, each mapper node evaluates Multiply on its locally stored file and generates key-values pairs for the reducer. The reducer simply computes the Sum step by adding all values with the same key and sending them back to the user. The user receives \( \frac{B}{k} \) values of size \( k \) from the reducers and decrypts to get \( file_x \).
\[
\begin{align*}
v & \leftarrow \{\} \text{ for } i = 1 \text{ to } N \text{ do} \\
& \quad \text{if } i = x \text{ then} \\
& \quad \quad v_i \leftarrow \mathcal{E}(1) \\
& \quad \text{end} \\
& \quad v_i \leftarrow \mathcal{E}(0) \\
& \text{end}
\end{align*}
\]

**Algorithm 1:** User function \( \text{GenQuery}(N, x) \)

\[
\begin{align*}
\text{for } i = 1 \text{ to } N \text{ do} \\
& \quad c \leftarrow B_i \cdot v_i \\
& \quad \text{Emit}(i, c) \\
& \text{end}
\end{align*}
\]

**Algorithm 2:** Server function \( \text{Map}(\text{file, } v) \)

\[
\begin{align*}
total & \leftarrow 0 \text{ for } i = 1 \text{ to } N \text{ do} \\
& \quad total \leftarrow total + v_i \\
& \text{end} \\
& \text{Emit}(\text{key, total})
\end{align*}
\]

**Algorithm 3:** Server function \( \text{Reduce}(\text{key, } v) \)

### 2.3.2 Encryption Scheme

Since the map phase of our protocol involves multiplying every piece of the data set by an encrypted PIR value, it is important that we choose an efficient cryptosystem. Traditional additively homomorphic cryptosystems, such as Paillier’s, use some form of multiplication as their homomorphism. That is, for elements \( a \) and \( b \), \( \mathcal{E}(a) \cdot \mathcal{E}(b) = \mathcal{E}(a + b) \). Since our map phase consists of multiplying ciphertexts by unencrypted scalars, we would have to do exponentiation of a ciphertext. PIRMAP, and all PIR schemes, must compute on the whole data set, so this step would be computationally intensive.

Similar to a recent finding [28], we mitigate this problem by using an efficient *somewhat homomorphic* encryption scheme. Here we analyze PIRMAP using one that relies on the trapdoor group assumption[23], but other schemes can be used as discussed in section 2.3.3. Fully homomorphic encryption schemes support an unlimited number of computations without increasing the size of the ciphertexts. In contrast, this scheme results in ciphertexts which grow in size by \( O(\log N) \) bits for \( n \) additions. In return for this size increase, we can have an encryption scheme where the additive homomorphism is integer addition. This scheme encrypts \( n \) bits with security parameter \( k > n \) as follows:

**KeyGen(1^k):** Generate a prime \( m \) of \( k \) bits and a random \( b < m \); \( b \) and \( m \) are secret.

**Encrypt \( \mathcal{E}(x) = b \cdot (r \cdot 2^N + x) \mod m \), for a random \( r \)**

**Decrypt \( \mathcal{D}(c) = b^{-1} \cdot c \mod m \mod 2^N \)**
This encryption has the desired homomorphic property $E(a) + E(b) = E(a + b)$. This scheme is somewhat homomorphic, because it cannot support an unlimited number of additions. When two ciphertexts $c_1$ and $c_2$ are added, you can express the sum as

$$b \cdot (r_1 \cdot 2^N + x_1) + b \cdot (r_2 \cdot 2^N + x_2) = b \cdot ([r_1 + r_2] \cdot 2^N + x_1 + x_2).$$

As $m$ remains secret, note that the cloud performs addition as integer addition, not modulo $m$. If the inside term $(r_1 + r_2) \cdot 2^N + x_1 + x_2$ exceeds $m$ and “wraps around”, then it will not be decrypted correctly, because application of the modulus will cause a loss of information. The modulus $m$ must be chosen large enough to support the number of additions expected to occur. To support $t$ additions, $m$ should be increased by $\log t$ bits. Additionally, each scalar multiplication can be thought of as up to $2^N$ additions, meaning that the size of $m$ must be doubled for each supported scalar product. For our PIR scheme, $m$ must be chosen to be $O(2k + \log N)$ to support the required homomorphic operations.

In return for the reasonable increase in ciphertext size caused by the larger modulus (about 300% in our evaluations in Section 2.4), we are able to do very efficient computations over the encrypted data. Additionally, encryption is equivalent to only two multiplications, an addition and a modular reduction, while decryption is one multiplication and a reduction. This compares very favorably with other homomorphic encryption schemes, such as Paillier, requiring a modular exponentiation.

With our encryption scheme defined, we can now express more precise computational complexities for the protocol. Our previous complexity was parameterized over $M(k)$ and $A(k)$, the complexity of scalar multiplication and addition, respectively. For our encryption, addition is simply regular integer addition. Since each ciphertext is at most $2k + \log N$ bits long, addition is $O(2k + \log N)$. We can do scalar multiplication as integer multiplication as well. Integer multiplication can be done for $m$ bits in $O(m \log m \log \log m)$ [29], so $M(k) = O(2k + \log N \log(2k + \log N) \log(\log(2k + \log N)))$. The complexity is then dominated by the multiplication cost and results in:

$$O(n \cdot \frac{B}{k} \cdot (k + \log N) \log(\log(k + \log N))) \quad (2.1)$$

$$= O((n \cdot B + N \cdot \log N) \log(k + \log N) \log(\log(k + \log N))) \quad (2.2)$$

If $B > N$, then $B > \log N$, and we can simplify to $O(N \cdot B \cdot \log(k + \log N) \cdot \log(\log(k + \log N)))$. Additionally, $k$ has to be much larger than $\log N$, otherwise the server has the resources to find key $(m, k)$ by brute force. This allows us to simplify to $O(N \cdot B \cdot \log k \cdot \log(\log k))$. 


2.3.3 Privacy Analysis

PIRMAP inherits privacy properties of the work it is based on, i.e., Lipmaa [21] and the PIR variant by Trostle and Parrish [23]. In the following, we sketch our privacy rationale.

PIRMAP is privacy-preserving, iff an adversary (the cloud) cannot guess, after each query, with probability greater than $1/N$, which file was retrieved by the user after an invocation of the protocol. There are two pieces of information that the adversary has access to: the set of uploaded files and the vector $\vec{v}$ of PIR values. The uploaded files are independent of any encryption used in the PIR protocol and can be efficiently simulated by the adversary. Therefore, privacy depends only on $\vec{v}$.

Vector $\vec{v}$ contains many encryptions of “0” and one encryption of “1”. The problem of determining which file was selected is then equivalent to distinguishing between encryptions of “0” and encryptions of “1” in the underlying encryption. However, the scheme we use is provably secure against distinguishing under the Trapdoor Group Assumption [23]. Consequently, PIRMAP preserves user privacy.

Security of Trostle-Parrish PIR  Unfortunately, after our experiments were ran and work was published [30], it was found that the trapdoor group assumption is vulnerable under lattice-based attacks[26]. Fortunately, subsequent work has confirmed our main result, that modern homomorphic encryption can make PIR practical. Doröz et al. [31] have shown that NTRU can be used in a similar manner to good effect, and there are other recent advances in homomorphic encryption that may be even faster[32, 33]. We have implemented the encryption scheme of Coron et al. [32] which uses the more standard Approximate GDC Assumption and present some results using that scheme in 3.3. These results show that, although not quite as efficient as the scheme by Trostle and Parrish, other encryption schemes can be used which are much faster than Paillier and which allow for significantly improved efficiency with PIR.

2.4 Evaluation

We have evaluated the performance of our scheme in three contexts: a local “cloud” (a single server with multiple CPUs), a commodity laptop, and Amazon’s EC2 cloud using Elastic MapReduce [17]. We have implemented PIRMAP in Java for standard Hadoop MapReduce version 1.0.3 and the source code is available for download [22].
2.4.1 Setup

Local We used a local server to prototype and debug our application and to do detailed timing analysis requiring many runs of MapReduce. This server, running Arch Linux 2011.08.19, has a dual 2.4 GHz quad-core Xeon E-5620 processor and 48 GB of memory. Based on specs and benchmark results, this local server is closest to an “EC2 Quadruple Extra Large” instance, which has dual 2.9 GHz quad-core Xeon X-5570 processors and 24 GB of memory.

We have measured the time needed for PIR queries, i.e., the time to upload PIR vector \( \vec{v} \) plus the time to process the query and download the result. Using Amazon’s standard cost model, we have calculated the price of each PIR query as the amount of money required to run the query on one of the above EC2 instances (for the same amount of time it took to run locally) plus the bandwidth cost of downloading the results [17]. Since uploading data to Amazon is free, this does not add any additional cost. To put our measurements into perspective, we have also evaluated time and cost of two other, hypothetical, PIR protocols. We have implemented a Baseline, which does not perform any cryptographic operations and merely “touches” each piece of data through the MapReduce API. This measure shows the theoretical lower bound of computation and time required for any PIR scheme that uses MapReduce, independent of the encryption and exact PIR method used.
To highlight the advantage of computational PIR, we have also included the time and cost required for the “trivial” PIR scheme. The trivial scheme is one where the user downloads the entire data set and simply discards the files he is not interested in. This is very bandwidth intensive, but computationally lightweight. Sion and Carbunar [16] conjecture this trivial PIR to be the most cost effective in the real-world. We have calculated the cost based on the amount that Amazon charges to download the corresponding amount of data and the time based on a 11.28 Mbps connection, an average as reported by Nasuni [34]. Note that we generally do not count the cost for long-term storage of data at Amazon. Although potentially significant for large amounts of data, the user has to pay for this regardless of whether he wants queries to be privacy-preserving or not. PIRMAP does not increase the amount of storage in Amazon.

Laptop We also ran our implementation on a 2012 MacBook Pro with a 2.3 Ghz i7, 8 GB of RAM and an NVIDIA GeForce 650M. The purpose of this test was to compare query time with related work, and the results are shown in Table 2.4. The Aguilar scheme takes advantage of GPU resources, so we chose a machine with comparable graphics and GPU resources (unlike the above server, which had no discrete graphics). This represents the performance you would get on a commodity machine and also shows that query generation is not very taxing. To compare with Lipmaa [21], we used the command openssl speed rsa to determine how many RSA private key operations can be computed per second. Each scalar multiplication in that scheme is equivalent to one modular exponentiation, or one RSA private key operation as required by Lipmaa [21]. This estimation is quite generous for Lipmaa [21], because it does not include disk access or homomorphic additions (modular multiplication), but we believe it is close due to the time being largely dominated by exponentiations.

Amazon To demonstrate the scalability of PIRMAP, we have also evaluated it on Amazon’s Elastic MapReduce cloud. Amazon imposes a maximum limit of 20 instances per MapReduce job by default. In keeping with this restriction, we used 20 instances denoted by Amazon as “Cluster Compute Eight Extra Large”, which each have dual eight-core Xeon E5-2670 processors and 64 GB of RAM.

<table>
<thead>
<tr>
<th></th>
<th>5 GB</th>
<th>10 GB</th>
<th>15 GB</th>
<th>20 GB</th>
<th>25 GB</th>
<th>30 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lipmaa</td>
<td>1852</td>
<td>3704</td>
<td>5508</td>
<td>7312</td>
<td>9116</td>
<td>10920</td>
</tr>
<tr>
<td>Aguilar-Gaborit</td>
<td>4</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>PIRMAP</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
2.4.2 Results

Time and total cost: Figures 2.1 to 2.4 show our evaluation results. Figures 2.1 and 2.2 show the local server evaluation, while figures 2.3 and 2.4 show evaluation with Amazon. In each figure, the x-axis shows the total amount of data stored at the cloud, i.e., number of files $N$ times file size $B$. The y-axis shows either the time elapsed (for the whole query from the time the user submits the query until MapReduce returns the result back) or the cost implied with the query. In all four graphs, we scale the y-axis logarithmically, and in figures 2.3 and 2.4 we also scale the x-axis logarithmically. Each data point represents the average of at least 3 runs. Relative standard deviation was low at $\approx 5\%$.

To verify the impact of varying file sizes $B$, for our local evaluation, we show results with file sizes of 1 MB and 3 MB, attaining approximately equal runtime in both cases. This is to be expected because, in each data point, we have fixed the size of the database so varying retrieval sizes merely reshapes the matrix without changing the number of elements in it. The execution is dominated by the scalar multiplications that occur during the map phase, and the same number of those is required no matter the dimension of the matrix.

Our evaluation shows that PIRMAP outperforms trivial PIR in both time and cost by one order of magnitude. We also compare performance to a baseline PIR protocol implemented in MapReduce where the mappers simply read and ignore any rows that are not requested and return the single row that was. Any MapReduce-based PIR protocol will require at least this amount of computation. Compared to this theoretical, yet unrealistic optimum, PIRMAP introduces only 20% of overhead in the case of Amazon. Locally, we experience slightly larger overhead of 100%. This is because executing on Amazon has a much higher “administrative” cost due to the higher number of nodes and more distributed setting. These results indicate not only PIRMAP’s efficiency over Trivial PIR and Baseline, but also its real-world practicality: in a small database comprising 10,000 files of size 1 MB each (10 GByte), a user can retrieve a single record in $\approx 3$ min for only $\approx $0.03. In a huge data set with 1,000,000 files, a single file can be retrieved in $\approx 13$ min for $\approx $14. In scenarios where it is necessary to retrieve data in a fully privacy-preserving manner, we conjecture that this to be acceptable.

Although a comparison with related research is not straightforward (as PIRMAP targets a very special scenario), we put our results into perspective with those of Aguilar and Lipmaa. We show that, while Lipmaa’s scheme has very good communication complexity in all cases, it is completely impractical due to the enormous amount of computation needed to respond to queries. We also show that our scheme is comparable with that of Aguilar, in terms of computation, and beats it in communication cost by a significant margin.

Query Generation and Decryption: Due to the efficiency of the encryption in PIRMAP, PIR query generation is very fast. One ciphertext (element of $\vec{v}$) is generated for each file in the
cloud, so the generation time is directly proportional to the number of files. We omit in-depth analysis, but in our trials on our commodity laptop running on a single core it takes about 2.5 seconds per 100,000 files in the cloud. Decryption is slightly more expensive than encryption, but we still managed, on the same machine, to decrypt approximately 3 MB per second. We conclude this overhead to be feasible for the real-world.

**Bandwidth:** PIRMAP introduces bandwidth overhead, 1.) to upload $\vec{v}$, and 2.) to download the encrypted version of the file. For security, we set $k = 2048$ bit, so each of the $n$ elements of vector $\vec{v}$ has size 2048 bit. For a data set with 10,000 files (10 GByte), this requires the user to upload $\approx 2.5$ MByte. As this can become significant with larger number of files, we suggest to then use the optimization in Section 2.3, especially for constrained devices.

### 2.5 Conclusion

Retrieval of outsourced data in a privacy-preserving manner is an important requirement in the face of an untrusted cloud provider. PIRMAP is the first practical PIR mechanism suited to real-world cloud computing. In the case where a cloud user wishes to privately retrieve large files from an untrusted cloud, PIRMAP is communication efficient. Designed for prominent MapReduce clouds, it leverages their parallelism and aggregation phases for maximum performance. Our analysis shows that PIRMAP is an order of magnitude more efficient than trivial PIR and introduces acceptable overhead over non-privacy-preserving data retrieval. Additionally, we have shown that our scheme can scale to cloud stores of up to 1 TB on Amazon’s Elastic MapReduce.
Chapter 3

Combining PIR and Oblivious RAM

As noted previously, there are traditionally two ways to hide a user’s access pattern, given a single server/cloud: Oblivious RAM (ORAM) [3] and Private Information Retrieval (PIR) [4]. The traditional approach taken by ORAM is to arrange the data in such a way that the user never touches the same piece twice, without an intermediate “shuffle” which erases the correlation between block locations. ORAMs have historically featured low amortized communication complexity and did not require any computation on the server, but occasionally the user was required to download and reshuffle the entire database. This could become impractical in cloud scenarios, especially if the user is a low-powered or communication-constricted device.

Private Information Retrieval, in contrast with ORAM, hides the target of each individual query, independent of all previous queries. This can be accomplished by using a homomorphic encryption which the server uses to operate over the entire database, selecting out the block of data that the user has requested. The user generates encrypted requests and sends them to the server. Since PIR does not try to hide a sequence of accesses, but each access individually, the amortized cost is equal to the worst-case cost. Unfortunately, the requirement that the server computes over the entire database for each query is often impractical, especially for large databases.

In this chapter, we present Path-PIR, a new ORAM construction combining ORAM and PIR, that overcomes the individual drawbacks of each of the two approaches. Path-PIR’s strategy is to augment the ORAM by Shi et al. [5] using techniques from PIR. It has been shown that PIR can be quite efficient when the block size is large relative to the number of elements in the database [30]. Since the tree-based ORAM of Shi et al. [5] is composed of many buckets, each of which has only a small number of elements, we are able to take advantage of PIR’s better worst-case communication guarantees, while at the same time using the tree structure to limit the portion of the database which is subject to expensive PIR operations.
TABLE 3.1: Communication complexity of Path-PIR and related constant-memory schemes. Here, $N$ is the ORAM capacity, e.g., the number of files, $B$ is the bit-length of each file, and $k$ is the security parameter. Latency is the amount of communication before the client has access to data. The “practical” setting is $B \geq 100$ KB and $N < 2^{35}$.

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Worst-Case</th>
<th>Practical Worst-Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shi et al. [5]</td>
<td>$O(B \cdot \log^2 N)$</td>
<td>$O(B \cdot \log^3 N)$</td>
<td>$O(B \cdot \log^3 N)$</td>
</tr>
<tr>
<td>Kushilevitz et al. [6]</td>
<td>$O(B \cdot \log \log N)$</td>
<td>$O(B \cdot \log \log N)$</td>
<td>$O(B \cdot \log^3 N)$</td>
</tr>
<tr>
<td>Path-PIR Linear</td>
<td>$O(k \cdot \log N + B)$</td>
<td>$O(k \cdot \log^2 N + B \cdot \log^2 N)$</td>
<td>$O(B \cdot \log N)$</td>
</tr>
<tr>
<td>Path-PIR FHE</td>
<td>$O(k + B)$</td>
<td>$O(k \cdot \log N + B \cdot \log N)$</td>
<td>$O(k + B)$</td>
</tr>
<tr>
<td>Optimal</td>
<td>$O(\log N + B)$</td>
<td>$O(\log N + B)$</td>
<td>$O(\log N + B)$</td>
</tr>
</tbody>
</table>

Additionally, we explore the notion of an ORAM’s latency, the amount of communication required before the client has access to their data. This is important, because many modern ORAM constructions involve an initial query, which returns the data, and a more expensive “book keeping” protocol which insures the integrity and obliviousness of the data structure. Low latency is a highly desirable feature, since the reorganizing and shuffling of the data structure can happen in the background after the client receives their data. We also note that low latency can be very useful in certain client settings with restrictive data limits, for example, smart phones.

Our Path-PIR scheme is especially suited to databases with large block sizes, an important setting in the real world that has not been thoroughly explored. For example, in medical applications, the size of each patient record (block size) can be quite large, due to medical images, test results, etc.

This chapter makes the following contributions:

1. Path-PIR, an ORAM construction which uses a combination of PIR and tree-based ORAM techniques to achieve good performance while maintaining constant client memory. Specifically, in a database that stores a total of $N$ files (entries), and each file is of bit length $B$, Path-PIR reduces communication from $O(B \cdot \log^3 N)$ to $O(\log^3 N + B \cdot \log^2 N)$. Path-PIR is especially efficient in practical real-world settings where $B \geq 100$ KB and $N < 2^{35}$, i.e., total databases of up to 3 PB size.

2. an improvement to Path-PIR which allows for optimal latency (the amount of communication spent before the user has access to the requested data) in retrieving blocks of size $B > O(\log^2 N)$.

3. a real-world implementation of Path-PIR, along with an evaluation performed on Amazon’s public EC2 cloud. Our evaluation shows that the additional computation imposed by PIR is outweighed by the significant data transfer savings. We show that Path-PIR allows significantly faster and cheaper operations than previous constant client memory constructions. The source code is available for download [22].
3.0.1 Motivation

Stefanov et al. [8] have shown that a tree-based construction can achieve $O(B \cdot \log N)$ amortized and worst-case complexity. However, this is achieved only with either linear client memory complexity or with square-root client memory complexity at the cost of additional communication complexity. Achieving constant client memory is an important requirement, because it allows ORAM applications for constrained devices such as smart phones and embedded systems, where memory may be limited. Unlike Shi et al. [5], the client memory required by Stefanov et al. [8] is dependent on the block size; for example, a 1 TB database of 1 MB files consumes approximately 800 MB of client memory in the square-root construction of Stefanov et al. [8]. It is unrealistic to expect 800 MB of free client memory for many real-world applications. Even in the linear client memory setting, Shi et al. [5] requires only 4 MB. This difference is caused by the fact that the Shi et al. [5] scheme has client memory independent of the block size, while Stefanov et al. [8] needs a block cache which can be very large for large block sizes.

For tree-based schemes, the size of the client memory in the linear setting is important, because it will govern how many recursive steps are needed in the constant-memory setting. Since linear-memory requirements are, for practical sizes, very low, we will see later that reducing to constant-memory requires only a small number of recursive steps (typically one).

One significant down-side to schemes which contain a block cache [5, 7, 8] is that the cache load can only be estimated empirically. If the size of the cache is chosen to be too small, and the actual cache needed during execution exceeds that size, then ORAM can only abort and lose its integrity. Even in devices with enough memory to allow for such schemes, it may be a significant burden on the client compared with a scheme that requires only a small constant memory.

Additionally, in order to make an ORAM available to multi-users, the client memory must be stored on the server between data accesses [35]. In such a situation, the client must include the size of their local memory in the communication cost of each request, significantly hampering the efficiency of schemes with non-constant memory requirements. This technique can be used with Path ORAM [7] to allow for constant long-term memory in exchange for more expensive data access, but clients still require non-constant transient memory in order to execute queries. This is a fundamental difference between Path ORAM and other related tree-based ORAMs.

3.1 Path-PIR’s Hybrid Construction

As usual, let $N$ denote the capacity, i.e., maximum number of blocks that can be stored in a database $D = \{d_0, \ldots, d_{N-1}\}$ at one time. We assume that all blocks are of equal size, and let $l$
denote the size of each block in bits. As in all tree-based ORAMs, we assume that $B > c \cdot \log N$ for some $c > 1$.

Although the complexity of constant client memory ORAM is only $O(B \cdot \log^3 N)$, a large value $B$ will render a ORAM scheme impractical in practice. Our aim is to modify the scheme so that it is practical for all large value $B$, while maintaining constant memory complexity on the client side.

### 3.1.1 Oblivious Outsourced Storage

Our hybrid construction will be different from traditional ORAM in one important way: it requires computation on the server side. Because of this, we require a slightly different security definition for a construct we call Oblivious Outsourced Storage.

**Definition 3.1.** Let

$$\bar{y} := \{(op_1, a_1, data_1), \ldots, (op_M, a_M, data_M)\}$$

be a sequence of data requests of length $M$, where $op_i$ is either read or write, $a_i$ is the address targeted by that operation, and $data_i$ is either the data to be written, if $op_i = \text{write}$, or $\perp$ if $op_i = \text{read}$.

Let $s$ be the security parameter, and $\text{Trace}_s(\bar{y})$ be the transcript of all messages sent between the client and server during the execution of an Oblivious Outsourced Storage protocol $\Pi(s)$ on $\bar{y}$. We say that $\Pi(s)$ is secure if, for any two data access patterns $\bar{y}$ and $\bar{z}$, and any probabilistic polynomial time adversary $A$,

$$|Pr[A(\text{Trace}_s(\bar{y})) = 1] - Pr[A(\text{Trace}_s(\bar{z})) = 1]| \leq \epsilon(s).$$

This definition is compatible with the traditional ORAM security definition, i.e., any secure ORAM will also be a secure OOS (since the data accesses induced by the server are part of the trace). However, it also incorporates the notion of server computation by stipulating that the entire trace must be indistinguishable instead of simply the access pattern on the server. The traditional ORAM definition does not need this requirement because there are no other interactions in the trace besides the access pattern. In our case, however, the trace includes PIR vectors and additional ciphertexts, so the definition must be modified slightly.
3.1.2 Private Information Retrieval

Since PIR has communication complexity which can be very efficient for retrieving large blocks, our goal will be to replace the bucket ORAMs in the previously described tree-based scheme with PIR queries to obtain better overall performance. Our approach in Path-PIR is to create a “PIR-bucket” to replace the bucket ORAMs at each node in the tree. However, it is not sufficient to simply replace the ORAM buckets with PIR, because buckets must have the ability to add and change blocks in order to support all the necessary ORAM operations. Therefore, in addition to standard PIR reading, we also need an equivalently secure writing protocol called “PIR-writing”. To begin, we will briefly define PIR and discuss relevant details of the PIR protocol we will be using.

**Definition 3.2.** A Private Information Retrieval protocol is a set of interactions between a user and a server comprised of the following functions:

- **PrepareQuery**\((x)\): Given a private input \(x \in \{1, 2, \ldots, N\}\), the user generates a query which is designed to retrieve the block with index \(x\) from the server.

- **ExecuteQuery**\((q)\): The server receives query \(q\) prepared by the user and executes it over the database \(D\). Here, \(D\) is a vector of \(N\) entries each of length \(l\). The response, consisting of the encoded requested block, is sent back to the user.

- **DecodeResponse**\((r)\): The user receives the server’s response to its query and decodes it to retrieve the requested block.

Along the same lines of “obliviousness” in ORAM, we define security (“Privacy”) for PIR.

**Definition 3.3** (Privacy). A Private Information Retrieval protocol is secure, if for any PPT adversary and any two indices \(y\) and \(z\), the corresponding queries \(Q = \text{PrepareQuery}(y)\) and \(Q' = \text{PrepareQuery}(z)\) are computationally indistinguishable.
We consider only single-server, computationally-secure PIR protocols, as multi-server schemes would not be appropriate for our setting. Though cloud providers may make use of many servers, they are all controlled by the same entity and may freely collude. It may be an interesting problem to consider multiple, competing cloud providers as a multi-server setting, but we leave the study of such a setting to future work.

In contrast with ORAM, PIR does not require keeping a state in between queries. Consequently, it can also be used to retrieve data from a public, unencrypted database. Since PIR protocols are stateless, each invocation of the protocol causes the server to perform $O(B \cdot N)$ computation. At a minimum, the server must “touch” each of the blocks in the $N$-capacity database or it could learn which blocks were not chosen by the user.

We have shown in chapter 2 that linear PIR can be done relatively efficiently if the block size is large in relation to $N$. However, PIR only works in one direction. ORAM also requires the ability to write to the database. Therefore we must introduce a compatible method for writing.

### 3.1.3 PIR-Writing

We define PIR-writing [36] as follows:

**Definition 3.4.** A PIR-writing protocol is a set of interactions between a user and a server comprised of the following functions:

- **PrepareWrite**$(x,y)$: Given a private input $x \in \{0, 1, ..., N\}$, the user generates a query which is designed to update block at index $x$ on the server with the new value $y$.

- **ExecuteWrite**$(q)$: The server receives query $q$ prepared by the user and executes it over the database, updating the corresponding block to its new value.

We stress that, in contrast to PIR, PIR-writing cannot be performed on unencrypted databases. As with ORAM, if the database was unencrypted, the server would learn immediately which record was changed. Still, PIR-writing has one interesting feature which is not subsumed by ORAM: it is also stateless. PIR-writing only requires a long-term key. In contrast, ORAM, even under constant user memory, requires state to be updated with each operation.

#### 3.1.3.1 Linear

Path-PIR’s linear PIR protocol above can be adapted to a PIR-writing protocol in a straightforward manner. If, instead of $D$, the server holds $C = \{\mathcal{E}(d_1), \ldots, \mathcal{E}(d_N)\}$, the protocol runs as follows:
Figure 3.2: PIR-Writing using the linear scheme. The cross product of the request vector (size $n$) and the change value (size $l$) is computed and added to the database.

1. PrepareQuery($x, y$) – The user generates a vector $Q = \{q_1, \ldots, q_n\}$ where $\forall i \neq x : q_i \leftarrow E(0)$ and $q_x \leftarrow E(1)$. Additionally, the user calculates $y' = y - d_x$ and returns the query $(Q, y')$.

2. ExecuteQuery($q$) – The server computes the cross product $\Delta C = y' \times Q$ and adds it to $C$ componentwise.

As before, multiplying by encryptions of zero will result in encryptions of zero, meaning that every block not being updated has an encryption of zero added to it which corresponds to a re-encryption. The single block being updated has an encryption of $y'$ added to it, resulting in a new value of $y$. This protocol requires that the user knows the current value of $d_x$, but this can be accomplished with a prior execution of PIR.

An additional problem with this protocol is that the server learns $y'$, the difference between the old value of $d_x$ and the new value. One might try to set $q_i = y' - y$ to get around this, but then the size of each encryption becomes $O(B)$, and we lose any benefit from using PIR. If, however, the user first encrypts the blocks with an IND-CPA encryption before applying the homomorphic encryption, the server sees only a difference between the two ciphertexts. This is equivalent to seeing two ciphertexts $((c_1, c_2) \oplus c_1) \Rightarrow (c_1, c_2)$, which gives the adversary no information under an indistinguishable encryption.

3.1.4 Replacing internal ORAM buckets with PIR

For the internal ORAM buckets, as stated above, we only need to provide a PIR capable of performing ReadAndRemove and Add, and that allows for a non-contiguos identifier space. This is because the bucket will be storing “sparse” identifiers, i.e., there are $N$ possible block identifiers and a random $O(\log N)$ subset of them will be in any given bucket. In order to
support the Add operation and the “remove” part of ReadAndRemove, any PIR construction requires also PIR-writing. From a high level perspective, our idea in Path-PIR is to implement ReadAndRemove and Add with one invocation of PIR and PIR-writing, respectively. Again, let \( n \) designate the capacity of a bucket and \( N \) the capacity of the entire ORAM. We further assume that the user has an IND-CPA additively homomorphic encryption scheme \((E, D, K)\), e.g., Paillier, and an IND-CPA symmetric encryption scheme \((E', D', K')\), e.g., AES-CBC with random IVs. We will first show how to construct a basic PIR bucket and then discuss additional improvements that can be made and interesting properties that arise from it.

It is sufficient to show that we can implement an oblivious bucket that supports ReadAndRemove and Add and that allows for a non-contiguous identifier space. By non-contiguous identifier space we mean that a bucket may hold \( n \) items, but the identifiers for those items may be from the set \( \{0, \ldots, 2^m\} \) with \( m > n \). This is required for the tree construction, because there are, overall, \( N \) elements in the ORAM, with \( N \) unique identifiers, and each bucket has capacity only \( n = \log N \). Therefore, there will be more possible identifiers than rows in the bucket. Standard PIR does not support a non-contiguous identifier space, as the “identifiers” are the row indices of each block in the database. We will overcome this in Path-PIR by using an encrypted map, stored on the server, which relates block identifiers to rows and allows us to use PIR with arbitrary identifiers.

Note that, in order to support the Add operation and the “remove” part of ReadAndRemove, any construction attempting this will also have to use a PIR-writing protocol to mask these operations. At a high level, the idea will be to implement ReadAndRemove and Add with one invocation of PIR and PIR-writing, respectively. We construct a store for the internal ORAM buckets meeting the above conditions for \( n \) blocks as follows:

### 3.1.4.1 Data storage

The server will store \( n \) tuples \((E_0(t), E_0(u), E(E_0(v)))\), where \( t \) is the leaf node that the block is moving toward, \( u \) is the block identifier and \( v \) is the actual data (“value”) of the block. If the row is empty (i.e., no block is currently stored there) then \( u \) is set to some canonical dummy value \( \perp \). The value for each block is stored double-encrypted so that we can use the PIR-writing protocols outlined above.

### 3.1.4.2 ReadAndRemove(x)

The user reads all the encrypted \( u \) values from the server (we will call these values the map) and learns in which row of the bucket block \( x \) resides in. If the requested block is present in this store at row \( i \), the user changes its \( u_i \) value to \( \perp \), reencrypts all \( u \) values with fresh randomness and
**Input:** Identifier \( x \) of block to retrieve

**Output:** Value of block \( x \) or \( \perp \) if block does not exist

begin

Read and decrypt the map \( U = \{u_1, \ldots, u_n\} \) from the server

\( i \leftarrow 0 \)

\( exists \leftarrow \text{false} \)

for \( j \in \{1, \ldots, n\} \) do

if \( u_j = x \) then

\( i \leftarrow j \)

\( u_j \leftarrow \perp \)

\( exists \leftarrow \text{true} \)

end

end

Reencrypt \( U \) and send back to the server

\( Q \leftarrow \text{PrepareRead}(i) \)

\( R \leftarrow \text{ExecuteRead}(Q) \)

if \( exists \) then

| return \( D'(D(\text{DecodeResponse}(R))) \)

else

| return \( \perp \)

end

Algorithm 4: ReadAndRemove

sends them back to the server. This marks the row as a “dummy” and effectively performs the “remove” part of ReadAndRemove. All rows in the map are reencrypted so the server does not learn which block the user was actually interested in. The user then executes PrepareRead\((i)\) and sends the results to the server. The server executes the query over \( V = v_1, \ldots, v_n \), returns the response, and the user decrypts it with \( D \) and \( D' \) to obtain the value for block \( x \). We do not have to remove or change the value \( v \) corresponding to the block that we are reading, but only change its identifier to \( \perp \). Future Add operations will simply overwrite the existing value.

### 3.1.4.3 Add\((x,y)\)

The user reads all encrypted \( u \) values from the server and selects an empty block \( i \) where \( u_i = \perp \). The user sets \( u_i = x \), reencrypts all \( u \) values and sends them back to the server. The user then runs PrepareWrite\((i,y)\) and the server executes the PIR-writing query over \( V \), changing the value in the \( i^{\text{th}} \) row to \( y \). Note that in order to calculate the query for PIR-writing, the user must already know the old value of the block. Therefore, there is an implicit PIR query that occurs as part of PrepareWrite, but it has the same communication complexity as the PIR-writing query.
**Input:** Identifier $x$ of block to add and value $y$ of said block

**Output:**

begin
  Read and decrypt $U = \{u_1, ..., u_n\}$ from the server
  $i \leftarrow 0$
  /* First, find an empty block in the bucket */
  for $j \in \{1, ..., n\}$ do
    if $u_j = \bot$ then
      $i \leftarrow j$
  end
  /* Mark that block with its new identifier */
  $u_i \leftarrow x$
  Reencrypt $U$ and send back to the server
  /* Read the existing block value */
  $Q \leftarrow \text{PrepareRead}(i)$
  $R \leftarrow \text{ExecuteRead}(Q)$
  /* Calculate the difference between the old and new values */
  $oldValue \leftarrow \mathcal{D}((\text{DecodeResponse}(R)))$
  $changeValue \leftarrow \mathcal{E}'(y) - oldValue$
  /* Write the change back to the bucket */
  $Q \leftarrow \text{PrepareWrite}(i, changeValue)$
  $\text{ExecuteQuery}(Q)$
end

**Algorithm 5:** Add

### 3.1.4.4 Complexity Analysis

The communication complexity for Path-PIR’s “PIR-bucket” is $O(n \cdot k + P(n))$, where $k$ is the block size of the additively homomorphic encryption, and $P(n)$ is the complexity of the underlying PIR protocol. For our linear scheme above, the communication complexity is $O(n \cdot k + B)$ so the overall communication complexity of the bucket is just $O(n \cdot k + B)$. Unlike ORAM, however, our PIR-bucket requires $O(n \cdot B)$ computation. When used in the larger ORAM construction, $n = \log N$, so this computation is quite reasonable as we will demonstrate in Section 3.3.

### 3.1.5 Improvements to the basic scheme

#### 3.1.5.1 Lower latency

An interesting property to consider for any ORAM is its data latency, that is the amount of data that is transferred before the client has access to the requested information. In our scheme, the client has access immediately after ReadAndRemove. Although Evict can be quite expensive, it can be executed in the background on the server without any user interaction, and the user
does not need to “wait” on it. At the tree level, the cost for a ReadAndRemove operation is 
\[ \log N \cdot P(\log N) \]. With \( P(n) = O(n \cdot k + B) \), this results in \( k \cdot \log^2 N + B \cdot \log N \). We can then 
save a factor of \( \log N \) by executing another PIR query over the results from each bucket in the 
path. As an example, if we know that the block we want is in row \( i \) of bucket \( j \) (from reading the 
headers) we show that the retrieval cost is only \( O(\log^2 N + B) \). We can send two PIR queries: 
the first selects the \( i^{th} \) row from every bucket, and the second selects the response from the \( j^{th} \) 
bucket (out of \( \log N \) buckets in the path). The overall latency is now \( k \cdot \log^2 N + B \), which is 
optimal for any retrieval within the constant factor \( k \). This leads to very low latency in practical 
situations (lower than any other previous work, even allowing for non-constant client memory).

3.1.5.2 Lower communication for Evict

Path-PIR’s default approach to Evict using its PIR-bucket is to simply execute a ReadAndRemove on the parent bucket and two Add operations on the children. This requires three 
PIR queries and two PIR-writing queries. Since the user knows which child node the block is 
going to be added to, it can simply execute a “dummy” PIR query over the other child node, 
where all the encryptions in the request vector are encryptions of zero. The same change value 
can then be used for both children, but the dummy request will simply result in an entire vector 
of zeroes and no change to the non-selected child bucket. With this modification, Path-PIR can 
coalesce the two reads and writes on the child nodes into one, saving a factor of \( 2 \cdot B \).

Fully homomorphic encryption (FHE). An interesting extension to our scheme would be to 
use a more powerful fully homomorphic encryption (FHE). The most communication intensive 
part of our scheme, which maintains a dependence between \( N \) and \( B \), is Evict. Unfortunately, 
since Evict is randomized, the user is required to send at least the random choices of buckets 
to the server, of which there may be many. However, if our adversary is honest-but-curious, 
then we can allow him to provide the randomness needed for the eviction process. This would 
eliminate any communication with the user and actually allow for evicts with a communication 
complexity of zero, given a fully homomorphic encryption scheme. The user can encode a 
circuit which evicts one block from a bucket to its children, and the server can run it on random 
buckets as normally chosen by the user. This would realize an ORAM with very close to optimal 
communication complexity \( (B \cdot \log N) \), since the read/write operations were already close to 
optimal, and eviction would cost nothing. It is not surprising that one can privately retrieve 
a block from a database with good communication complexity using FHE, since retrieval is 
equivalent to testing equality over encrypted bit-strings – and this can be performed quite easily. 
However, it is interesting that by using a tree construction we can achieve low communication 
complexity while only computing over a \( O(B \cdot \log^2 N) \)-sized fraction of the database. Any FHE-
based approach is likely to be restricted by the expensive ciphertext operations. Consequently, it 
is very helpful that computation only needs to be done over a small portion of the database with
each user operation. Unfortunately, fully homomorphic encryption is still too impractical to be used in this manner, but might become attractive in the future.

3.1.6 Summary: Complexity Analysis

Table 3.1 on page 29 compares the communication complexity of related work with Path-PIR. The last column shows the performance of each scheme in a setting with moderately large blocks ($B > 100$ KB) and practical sized databases of up to 3 PB. The two bucket based schemes perform better in this setting because the depth of recursion is limited to one (they lose a log $N$ factor). Additionally, Path-PIR performs especially well, since all the ciphertexts that need to be transferred are significantly less than the size of one block, so the communication is dominated by the $O(\log N)$ blocks which are transferred during Evict.

In conclusion, Path-PIR reduces the expensive communication complexity that depends on file length $B$ by a factor of $\log N$ using a simple “linear” PIR protocol. Although a reduction from $\log^3$ to $\log^2$ may look small, the total savings can be substantial in practice – as we will demonstrate by our experimental results presented in Section 3.3.

We observe in Path-PIR that, although the user needs to perform one eviction for each read or write operation, these evictions are not required to be performed immediately after the operation. The contribution of eviction is to keep buckets from overflowing, but the correctness and security of the ORAM remains independent of it. The user can actually conduct $O(\log N)$ data accesses without any evictions before the root node will overflow. Since ReadAndRemove and Add are very efficient, and the overwhelming majority of communication is consumed during Evict, this could be very useful when a user’s cost on communication may vary in different environments. For instance, a user with a cell phone may pay significantly more money for cellular data than WiFi data. In a practical implementation of Path-PIR, one could defer evictions while they are on expensive cellular data and choose to perform these operations later when they are on cheap WiFi. This allows for extremely low communication requirements while Evict operations are being deferred. Additionally, the size of the root bucket can be increased by any constant factor to allow for more deferred operations without effecting the overall complexity.

3.2 Security

It is relatively simple to show the security of our scheme. At a high level, we are just composing PIR and ORAM techniques which are secure individually, and we shall see that they remain secure together. The PIR elements that we use result in the transfer of many ciphertexts between the client and the server. However, these ciphertexts are freshly generated by the client for each query, so under IND-CPA security they should not give any information to the server.
First, we show that an individual bucket, created using PIR techniques, is secure.

**Lemma 3.5.** *Path-PIR’s construction of a PIR-bucket is a secure oblivious operation.*

**Proof.** There are three parts of each PIR-bucket operation: retrieval of the map, execution of the PIR query and execution of the PIR-writing query.

The map is oblivious, because it is encrypted and retrieved in its entirety for every operation. At the conclusion of an operation, the entire map is freshly reencrypted and sent back to the server. In that way, it is actually a “trivial” ORAM and therefore meets our security definition.

Next, we show that Path-PIR is insecure only, if the IND-CPA security of either the symmetric encryption or the additively homomorphic encryption is invalid. Let \( Q(\tilde{y}) = \{Q_1, \ldots, Q_N\} \), where \( Q_i = \{q_{i,0}, \ldots, q_{i,m}\} \), denote the set of all PIR and PIR-writing queries involved in the execution of a set of data accesses \( \tilde{y} \). Each \( q_{i,j} \) is an encryption of the plaintext \( p_{i,j} \) under our additively homomorphic IND-CPA encryption scheme. Similarly, let \( P(\tilde{y}) = \{P_1, \ldots, P_n\} \) denote the set of all plaintexts used in the construction of \( Q(\tilde{y}) \). Assume that Path-PIR is insecure, that is, there exists an adversary, \( A \), that can distinguish between two access patterns, \( \tilde{y} \) and \( \tilde{z} \).

We will construct an adversary, \( A' \) which can break the IND-CPA security of our homomorphic encryption.

\( A' \) runs \( A \) and obtains the two access patterns, \( \tilde{y} \) and \( \tilde{z} \), which \( A \) can distinguish. \( A' \) then simulates Path-PIR on the two access patterns, obtaining \( X_0 = P(\tilde{y}) \) and \( X_1 = P(\tilde{z}) \). \( A' \) submits \( X_0 \) and \( X_1 \) to the IND-CPA challenger and receives back \( C \), an encryption of one of the two plaintexts. He then sends \( C \) to \( A \) as the trace and \( A \) returns either \( y \) or \( z \) as the chosen access pattern. If \( A \) returns \( y \), then \( A' \) returns 0 to the challenger, otherwise returns 1. Thus, if \( A \) is able to distinguish between the two access patterns with probability \( 1/2 + \epsilon \), then \( A' \) will be able to distinguish between the two encryptions with the same probability.

Note that the trace will also include responses from the server, but those cannot reveal any additional information, because they are deterministically computed from the client’s requests.

We have now established that the construction of a PIR-bucket is a secure oblivious operation. \( \square \)

**Theorem 3.6.** *The basic version of Path-PIR is a secure Oblivious Outsourced Storage protocol.*

**Proof.** The security proof by Shi et al. [5] shows that the sequence of bucket accesses will be oblivious. As stated above, the standard definition of ORAM security is compatible with our OOS security definition, therefore the security of our scheme depends only on the PIR-bucket itself being oblivious, which is established by Lemma 1. \( \square \)
Theorem 3.7. The improved latency version of Path-PIR is a secure Oblivious Outsourced Storage protocol.

Proof. The only difference between the improved latency scheme and the original one is that a single PIR vector is “reused” for every bucket on a path in the tree. However, since PIR vectors are themselves indistinguishable for any two indices being accessed, “reusing” them does not provide any additional information and therefore, the protocol remains oblivious. If an adversary could distinguish in the case with just one request vector, he could equally distinguish that same vector in the original scheme by ignoring all other vectors.

Based on the oblivious secure protocol properties of PIR and ORAM, we have established that Path-PIR satisfies the Oblivious Outsourced Security properties.
3.3 Evaluation

In order to become deployable in a practical, real-world cloud setting, any ORAM protocol must be parallelizable. The only way to scale up in the cloud is to expand to more nodes and CPUs in the cloud’s data center. Fortunately, PIR as we have described is highly parallelizable. The scalar multiplication on each file can be evaluated independently, so Path-PIR can take advantage of up to $O(\log^2 N)$ independent CPUs.

Typically, public cloud providers such as Amazon, charge users for both communication/data transfer and CPU time [37]. As Path-PIR imposes additional computational requirements, the question is how the additional computational costs relate to the lower communication costs. We have implemented Path-PIR in Java and run simulations in Amazon’s EC2 cloud. Path-PIR’s source code is available for download [38]. As in the previous chapter, and other related work [28], we initially used the PIR scheme from Trostle and Parrish [23] because of its conceptual simplicity and efficient server computation (adding two PIR values is simply an integer addition). We chose security parameter $k = 2048$ as recommended by the authors.

As noted in chapter 2, this scheme was later shown to be insecure. We should note that other researchers have built upon the idea of computation-aided Oblivious RAM after Path-PIR was published [39] and constructed an ORAM with $O(1)$ online complexity which requires only XOR and not any homomorphic encryption at all [40]. We have subsequently implemented the encryption scheme of Coron et al. [32] and reran the experiments to show that our scheme still achieves an improvement over related work.

Figures 3.4(b), and 3.4(a) show both the time and cost of a query using both the original insecure encryption and the new scheme. Although computational cost increases by about 4x, the monetary cost remains quite low. This is due to the fact that computation is much cheaper than communication on current cloud platforms. We also emphasize that this cost could likely be reduced dramatically with optimized code or by an advancement in homomorphic encryption.

Setup. To benchmark our PIR protocols, we conducted our experiments using a single High-CPU Extra Large instance. One hour of CPU time with such an instance costs $0.58. To compare, Amazon charges $0.12 per GB transferred [37] (for the first 12 TByte). The worst-case communication cost of Shi et al. [5] and Path-PIR can be exactly computed based on $N$ and $l$. Related work requires no server computation, so we modeled cost based on communication alone. To estimate communication time (download/upload), we assume an 88 Mbps connection, in line with the maximum speed one would expect when transferring from Amazon S3 [34]. This is a very generous estimate, and our scheme compares even more favorably in bandwidth constrained environments. Computation time for Path-PIR is calculated by benchmarking on buckets of various sizes (using EC2).
Path ORAM We include comparison with Path ORAM [7] (even though it is not constant-memory) since it is currently the most efficient known ORAM protocol. We show that our scheme is very competitive with Path ORAM while requiring only constant client memory.

Figures 3.3(a), 3.4(b), and 3.4(a) show relative communication, time, and monetary cost per read/write operation for related work and Path-PIR. We consider databases of 1 MB blocks, with total size between 1 and 16 TB. As the block size increases, Path-PIR becomes more efficient relative to related work. Note that our scheme has even lower bandwidth requirements than Path ORAM and very competitive monetary cost.

Latency Figure 3.3(b) shows the extremely low cost of ReadAndRemove operations in our scheme. This latency property is important, because it represents the amount of communication necessary before the client has access to their data. The eviction, which takes up most of the communication, can be done in the background without user interaction. We are able to obtain extremely low communication requirements for this operation, since it requires transmitting only one full block. Even compared to the best related work in any memory setting, Path-PIR obtains vastly better latency.

3.4 Conclusion

Path-PIR demonstrates that integrating PIR into recent ORAM mechanisms provides better communication without incurring unreasonably large computational burden on the cloud. Our experiments confirm that Path-PIR’s cost savings from the lowered communication complexity are significantly higher than the cost of extra computation. Additionally, Path-PIR benefits from constant memory complexity and low latency that makes it especially conducive to constrained devices like cell phones or embedded systems.
A crucial aspect of ORAM schemes is their induced overhead. The choice to use the cloud is chiefly motivated by cost savings. If the overhead is enough that it negates any monetary advantages the cloud can offer, the use of ORAM will be impractical. Previous ORAM schemes have had a common, major drawback that has hindered real-world use. Due to eventually necessary “reshuffling” operations, their worst-case communication complexity was linear in the size of the ORAM. However, recent works on ORAM, e.g., by Shi et al. [5], Stefanov et al. [7] have proposed new tree-based ORAM schemes that have only poly-logarithmic worst-case communication complexity.

Unfortunately, the new tree-based approaches have exposed another barrier to the real-world adoption of ORAMs: the maximum size of the data structure must be determined during initialization, and it cannot be changed. This is not an issue in previous linear schemes, because the client always had the option of picking a new size during the “reshuffling”, being effectively a “reinitialization” of the ORAM. In tree-based ORAMs, though, a reinitialization ruins the sublinear worst-case communication complexity.

Resizability is a vital property of any ORAM to be used for cloud storage. One of the selling points of cloud services is elasticity, the ability to start with a particular footprint and seamlessly scale resources up or down to match demand. Imagine a startup company that wants to securely store their information in the cloud using ORAM. At launch, they might have only a handful of users, but they expect sometime in the long-term to increase to 10,000. With current solutions, they would have to either pay for the 10,000 users worth of storage starting on day one, even though most of it would be empty, or pay for the communication to repeatedly reinitialize their database with new sizes as they become more popular. Reinitializing the ORAM would negate any benefit from the new worst-case constructions. Additionally, one can imagine a company that is seasonal in nature (e.g., a tax accounting service) and would like the ability to downsize their storage during off-peak times of the year to save costs.
Consequently, the problem of resizing these new tree-based ORAMs is important for practical adoption in real-world settings. In light of that, we present several techniques for both increasing and decreasing the size of recent tree-based ORAMs to reduce both communication and storage complexity. We focus on constant client memory ORAM (Shi et al. [5]) since it is an interesting setting, especially for hardware-constrained devices and large block sizes or situations where multiple parties want to share the same ORAM and so need to frequently exchange the client state. We are able to show that, although the resizing techniques themselves are intuitive, careful analysis is required to ensure security and integrity of ORAMs. In addition, we show that it is nontrivial to both allow for sublinear resizing and maintain the constant client memory property of Shi et al. [5] ORAM.

The technical highlights of this chapter are as follows:

1. Three provably secure strategies for increasing the size of tree-based ORAMs, along with a rigorous analysis showing the impact on communication and storage complexity and security.

2. A provably secure method for pruning the trees to decrease the size of a tree-based ORAM, along with rigorous analysis showing that security and integrity of the data structures is preserved.

3. A new, tighter analysis for the Shi et al. [5] ORAM which allows for smaller storage requirements and less communication per query than previous work.

### 4.1 Resizable ORAM

The challenge behind resizing tree-based ORAMs is threefold:

1. Increasing the size of the tree will have an impact on the bucket size. A leaf node may become an interior node while increasing the ORAM, and vice versa in the decreasing case. The original analysis by Shi et al. [5] differentiates between interior and leaf nodes, while for resizing we will have to generalize the analysis to consider both cases at once.

2. For \( n > N \) elements, we must determine the most effective strategy of increasing the number of nodes to optimize storage and communication costs for the client.

3. Reducing the size of the tree is non-trivial, especially when targeting low communication complexity and constant client memory. A mechanism is required for moving elements from pruned nodes into other buckets in an oblivious, yet efficient way while still maintaining overflow probabilities.
4.1.1 Resizing Operations

To allow for resizing, we introduce two new basic operations by which a client can resize an ORAM, namely Alloc and Free:

- **Alloc**: Increase the size of the ORAM so that it can hold one additional element of size $B$.
- **Free**: Decrease the size of the ORAM so that it can hold one element fewer.

4.1.2 Security Definition

Resizing an ORAM should not leak any information besides the current number of elements. Thus, we need to augment the standard ORAM security definition by our resizing operations.

**Definition 4.1.1.** Let $\mathcal{Y} = \{(op_1, d_1, a_1), (op_2, d_2, a_2), \ldots, (op_M, d_M, a_M)\}$ be a sequence of $M$ operations $(op_i, d_i, a_i)$, where $op_i$ denotes a Read, Write, Alloc or Free operation, $a_i$ equals the address of the block if $op_i \in \{\text{Read, Write}\}$ and $d_i$ the data to be written if $op_i = \text{Add}$.

Let $A(\mathcal{Y})$ be the access pattern induced by sequence $\mathcal{Y}$. A resizable ORAM is secure iff, for any PPT adversary $\mathcal{D}$ and any two same-length sequences $\mathcal{Y}$ and $\mathcal{Z}$ where $\forall i \in [M]: \mathcal{Y}(i) = \text{Alloc} \iff (\mathcal{Z}(i) = \text{Alloc}) \land (\mathcal{Y}(i) = \text{Free}) \iff (\mathcal{Z}(i) = \text{Free})$,

$$\Pr[\mathcal{D}(1^\lambda, A(\mathcal{Y})) = 1] - \Pr[\mathcal{D}(1^\lambda, A(\mathcal{Z})) = 1] \leq \epsilon(\lambda),$$

where $\lambda$ is a security parameter, and $\epsilon(\lambda)$ a negligible function in $\lambda$.

For the sake of completeness, considering buckets in resizable ORAM as trivial ORAMs [3], all blocks are IND-CPA encrypted. Also, whenever a block is accessed by any type of operation, its bucket is re-encrypted block-by-block.

4.2 Adding

We begin by describing a naïve solution that will add a new level of $2N$ leaves when $n > N$. However, this already leads to a problem: when $n$ is only slightly larger than $N$, we are using twice as much storage as we should need. The second strategy, lazy expansion, will postpone creation of an entire new level until we have enough elements to really need it. In both the naïve and second solution, there are thresholds causing large “jumps” in storage space. As this can be expensive, we present a third solution dubbed dynamic expansion. This strategy progressively adds leaf nodes to the tree, thereby gradually increasing the tree’s capacity. This last strategy is particularly interesting, because it results in an unbalanced tree, requiring careful analysis to ensure low overall failure probability of the ORAM.
4.2.1 Tightening the bounds

Communication and storage complexities represent the core comparative factor between strategies, and both are dependent primarily on bucket sizes. Consequently, it is important to get a tight analysis for both interior and leaf bucket sizes. The original bounds for bucket sizes given by Shi et al. [5] are substantially larger than necessary. Therefore, as a first contribution, we give new, tighter bounds for interior and leaf node sizes.

4.2.1.1 Interior Nodes

We first address the size of interior nodes by using standard queuing theory. Let $I_i$ denote the random variable for the size of interior nodes of the $i^{th}$ level in the tree. For eviction rate $\nu$, we compute the probability of a bucket on levels $i > \log \nu$ having a load of at least $k$ (i.e., a size $k$ bucket overflows) as:

$$\Pr(I_i \geq k) = \nu^{-k}. \quad (4.1)$$

In [5], the eviction rate was chosen to be equal to 2 with an overflow probability equal to $2^{-k}$. However, if we adjust the bucket size to be $\frac{k}{\log(\nu)}$, the overflow probability is still $2^{-k}$, namely $\Pr(I_i \geq \frac{k}{\log(\nu)}) = 2^{-k}$.

This follows from Eq. 4.1 by replacing $k$ with $\frac{k}{\log(\nu)}$. Also, we can investigate the optimal value for the eviction rate $\nu$ in terms of communication cost. For $\nu = 4$, we obtain the same overflow probability as with $\nu = 2$, using buckets of half the size. The communication complexity does not change, as we are evicting twice as much, but with buckets of half the size. For larger eviction rates $\nu > 4$ the communication complexity becomes larger. Note that this also reduces the storage by a factor of 2. For $N$ elements stored in the ORAM, the probability that an interior node overflows during eviction computes as

$$\Pr(\exists i \in [\nu \cdot \log N] : I_i \geq \frac{k}{\log(\nu)}) = 1 - \Pr(\forall i \in [\nu \cdot \log N] : I_i < \frac{k}{\log(\nu)}) \quad (4.2)$$

$$= 1 - \prod_{i=1}^{\nu \cdot \log N} (1 - \Pr(I_i \geq \frac{k}{\log(\nu)})) \quad (4.3)$$

$$= 1 - (1 - 2^{-k})^{\nu \cdot \log N}.$$

In particular for $\nu = 4$, the optimal choice of the eviction rate,
The buckets that can overflow during an access are limited to those in the paths accessed during the eviction, i.e., $\nu \cdot \log N$ buckets accessed. Also, the number of buckets taken into account is actually $\nu \cdot \log N$ instead of $2\nu \cdot \log N$. This follows from the fact that for every parent, we write only one real element to one child. Consequently, per eviction and per level, only one child can overflow. Since the buckets can be considered independent in steady state, Eq. 4.3 follows directly from Eq. 4.2 [41].

Given security parameter $\lambda$, to compute the size of interior buckets, we solve the equation $2^{-\lambda} = 1 - (1 - 2^{-k})^{\nu \cdot \log N}$ to obtain $k = -\log (1 - (1 - 2^{-\lambda})^{\frac{1}{\nu \cdot \log N}})$. For example, to have an overflow probability equal to $2^{-64}$, $\lambda = 64$, $N = 2^{30}$, $\nu = 4$, the bucket size needs to be only 36 while Shi et al. [5] determined the bucket size be equal 72 for the same overflow probability. Moreover, since $N$, the number of elements in the ORAM, has a logarithmic effect on the overflow probability, the size of interior nodes will not change for large fluctuations of the number of elements $N$. For example, for $N = 2^{80}$, the interior node still has size 36 with overflow probability $2^{-64}$.

### 4.2.1.2 Leaf Nodes

Let $B_i$ denote the random variable describing the size of the $i^{th}$ leaf node. Thinking of a leaf node as a bin, a standard balls and bins game argument provides us the following upper bound

$$
\Pr(B_i \geq k) \leq \binom{N}{k} \cdot \frac{1}{N^k} \leq \frac{e^k}{k^k}.
$$

The second inequality follows from an upper bound of the binomial coefficient using Stirling’s approximation. For $N$ leaves, we have

$$
\Pr(\exists i \in [N] : B_i \geq k) = \Pr\left(\bigcup_{i=1}^{N} B_i \geq k\right) \\
\leq \sum_{i=1}^{N} \Pr(B_i \geq k) \\
\leq \frac{N}{e^{k-(\ln(k)-1)}}. 
$$
Note that in Eq. 4.4, we have used the union bound. Based on the same parameters as in the previous example, the size of a leaf node has to be set only to 28 to have an overflow probability equal to $2^{-64}$. To compute this result, one solves the equation $k = e^{W\left(\log_2{\frac{2^h N}{e}}\right)+1}$, where $W(.)$ is the product log function. While the size of the interior node can be considered constant for large fluctuations of $N$, the size of a leaf node should be carefully chosen depending on $N$. Every time the number of elements increases by a multiplicative factor of 32, we have to increase the size of the leaf node by 1 to keep the same overflow probability.

Note that for both interior and leaf node size computations, we do not take into account the number of operations (accesses) performed by the client. As with related work\[5\], the number of ORAM operations is typically considered part of security parameter $\lambda$. The larger the number of operations performed, the larger the security parameter has to be.

### 4.2.2 1st strategy: naive expansion

Let $N$ and $n$ respectively denote the number of leaf nodes and elements in the ORAM. The naïve solution is simply adding a new leaf level, as soon as the condition $n > N$ occurs. During eviction, when blocks are moved out of the old leaf nodes into the new ones, a random coin can be flipped and a bit added onto the tag of the blocks being moved down in order to maintain a uniform distribution of tags over the leaf nodes. The main drawback of this first naïve solution is the waste of storage which can be explained from two different perspectives. The first storage waste consists on creating, in average, more leaf nodes than elements in the ORAM. The second storage waste in the under-usage of the leaf nodes while they can hold more elements with a slight size increase. Our second strategy will try to get rid of this drawback.

### 4.2.3 2nd strategy: lazy expansion

This technique consists of creating a new tree level when the number of elements added is equal to $\alpha$ times the number of leaf nodes in the tree, where $\alpha$ is a constant greater than 1. For a $N$ leaves tree, the client is allowed to store up to $\alpha \cdot N$ elements in the ORAM without increasing the size of the tree. As soon as $n > \alpha \cdot N$, the client asks the server to create a new level of leaves with $2 \cdot N$ leaf nodes.

This lazy increase strategy is performed recursively. For example, if the size of the ORAM tree is now equal to $2 \cdot N$, then the client will work with the same structure as long as $\alpha \cdot N < n \leq \alpha \cdot 2 \cdot N$. Once $n > \alpha \cdot 2 \cdot N$, a new level of leaves with now $4N$ leaf buckets is created.

To be able to store more elements, our idea is to slightly increase the leaf bucket size. Therewith, we can keep the same overflow probability. Note the tradeoff between increasing the size of leaf
nodes and the communication complexity of the ORAM. To read or write an element in the ORAM, the client downloads the path starting from the root to the leaf node. If the size of this path (when increasing the size of the bucket) is larger than a regular ORAM tree with the same number of elements, then this technique would not be worth applying.

Gentry et al. [42] have shown that by increasing the leaf node size from $k$ to $\alpha + k$, we can reduce the storage overhead while handling more elements than leaf nodes. For $N$ leaf nodes, we can have up to $\alpha \cdot N$ elements. While Gentry et al. [42] chose $\alpha$ to optimize the storage cost for a given overflow probability, we instead target the computation of the value $\alpha$ to optimize communication complexity.

First, we determine a relation between the size $x$ of a leaf bucket and factor $\alpha$ for our 2nd strategy. Then, we compute the optimal value of $\alpha$ as a function of the security parameter $\lambda$, the size of the interior nodes, and the current number of leaves. To calculate the overflow probability, we focus on the worst case, when there are $\alpha \cdot N$ elements in an ORAM with $N$ leaves.

**Lemma 4.2.1.** Let $x$ denote the optimal leaf bucket size for the 2nd strategy. Then,

$$
\alpha = \frac{x}{e} \cdot \left( \frac{2^{-\lambda}}{N^x} \right)^{\frac{1}{x}}
$$

where $\lambda$ is the security parameter and $N$ the number of leaf nodes.

**Proof.** The situation is analogous to balls-and-bins where we are inserting uniformly at random $\alpha \cdot N$ balls into $N$ bins. The $i$th bin overflows if there are $x$ balls from $\alpha \cdot N$ that went to the same $i$th bin. The possible number of combinations is equal to $\binom{\alpha \cdot N}{x}$. By applying the upper bound inequality to the probability of the union of events (possible combinations), we obtain

$$
\Pr(B_i \geq x) \leq \left( \frac{\alpha \cdot N}{x} \right) \cdot \frac{1}{N^x} \\
\leq \left( \frac{e \cdot \alpha \cdot N}{x} \right)^x \cdot \frac{1}{N^x} \\
= \left( \frac{e \cdot \alpha}{x} \right)^x.
$$

Computing the union bound over all leaf nodes results in

$$
\Pr(\exists i \in [N] : B_i \geq x) \leq N \cdot \left( \frac{e \cdot \alpha}{x} \right)^x.
$$

In order to have overflow probability equal $2^{-\lambda}$ as previous work, we must verify that $N \cdot \left( \frac{e \cdot \alpha}{x} \right)^x = 2^{-\lambda}$ which is equivalent to $\alpha = \frac{x}{e} \cdot \left( \frac{2^{-\lambda}}{N^x} \right)^{\frac{1}{x}}$.

**Corollary 4.2.1.** Let $k$ denote the size of the interior node. The optimal communication complexity for the 2nd strategy is achieved iff the leaf bucket size $x$ equals
\[ x = \frac{k}{\ln 2} + \sqrt{k - 4 \cdot k \cdot \log \frac{2\cdot\lambda}{N}} \]

**Proof.** First, note that if \( N \) leaf nodes can handle \( \alpha \cdot N \) elements, the tree is flatter compared to the naïve solution where the tree will have height \( \log N \) instead of \( \log \alpha \cdot N \). However, the downside of the 2nd strategy is the leaf bucket size increase. In order to take the maximal advantage of this height reduction, we define the optimal leaf bucket size \( x \) that can have the best communication complexity compared to the naïve solution. Let \( C_1 \) and \( C_2 \) denote, respectively, the communication complexity needed to download one path for the first and second strategy. For an interior node with size \( k \) and a leaf bucket for the naïve strategy with size \( y \), the communication complexities \( C_1 \) and \( C_2 \) can be computed as

\[ C_1 = (\log \alpha \cdot N - 1) \cdot k + y \quad \text{and} \quad C_2 = (\log N - 1) \cdot k + x. \]

Wanting to improve on the naïve scheme, the best value of \( x \) for a fixed value of \( y, k \) and \( \lambda \) is the maximum value of the function \( f \) defined as

\[ f(x) = C_1 - C_2 = y - x + k \cdot \log \alpha. \]

The first derivative of \( f \) is \( \frac{df}{dx}(x) = x^2 + \frac{k}{\ln(2)} - x + k \cdot \log \frac{2\cdot\lambda}{N} \). This quadratic equation has only one valid solution for a non-negative leaf buckets size and \( 2^\lambda \gg N \). The only valid root for the first derivative is

\[ x = \frac{k}{\ln 2} + \sqrt{k - 4 \cdot k \cdot \log \frac{2\cdot\lambda}{N}}. \]

Once we have computed the optimal leaf node size, we can plug the result into Eq. 4.5 to compute the optimal value \( \alpha \). For example, for \( N = 2^{30} \) leaves, the size of the leaf bucket in the naïve strategy is \( y = 28 \), the size of the interior node \( k = 36 \). Applying the result of Corollary 4.2.1 outputs the size of the leaf bucket for an optimal communication complexity which is equal to \( x \approx 85 \). Applying the result of Lemma 4.2.1, we obtain \( \alpha \approx 15 \). The communication complexity saving compared to the naïve strategy is around 7% while the storage savings is a significant 87%.

One disadvantage of the 2nd strategy is the possibility of storage underutilization. Imagine the client stores \( \alpha \cdot N \) elements in the ORAM tree. When adding a new element, it will trigger the creation of a new leaf level, which is a waste of storage. For example, the client can have \( \alpha \cdot N + 1 \) elements in his ORAM tree, then performs a loop which respectively adds and deletes two elements. This loop will imply the allocation of an unused large amount of storage (in \( O(N) \)). Also, this loop implies leaf node pruning which is more expensive (in term of communication complexity) compared to leaf increasing as we will see in Section 4.3.
4.2.4 3rd strategy: dynamic expansion

To tackle the underutilization of storage problem in lazy expansion, we introduce a dynamic strategy. Instead of adding entire new levels to the tree, we will progressively add pairs of leaf nodes to gradually increase the capacity of the tree. This has the advantage of matching a user’s storage cost expectation: every time the ORAM capacity is increased, storage requirements increase proportionally. However, unlike our previous techniques, we are now no longer guaranteed to have a full binary tree. This implies a overflow probability recalculation of two different levels of leaf nodes.

Let us assume that we start with a full binary tree containing \( N = 2^l \) leaf nodes. Dynamic insertion results in the creation of two different levels of leaves. The first one is on the \( l \)th level while the other one is on the \((l + 1)\)th level. After ever \( \alpha \) insertion, a new pair of leaf nodes is added to the tree. In general, after adding \( \eta \cdot \alpha \) elements to an ORAM with an initial size of \( N = 2^l \), the number of leaves in the \( l \)th level is equal to \( N - \eta \) while the number of leaves in the \((l + 1)\)th level is equal to \( 2\eta \).

At this point, we must carefully consider how to tag new elements that are added to the tree. If we choose tags following a uniform distribution over all the \( N - \eta + 2 \cdot \eta = N + \eta \) leaves, we will violate ORAM security. An adversary will be able to distinguish with non-negligible advantage between two elements added before and after increasing the number of leaf nodes in the ORAM, as the assignment probabilities to (leaf) nodes will be different at varying points in the tree’s lifecycle.

An efficient solution to this problem is to keep the probability assignment of leaf nodes equally likely for all subtrees with a common root. We implement this approach by setting a leaf’s assignment probability in the \( l \)th level to \( \frac{1}{2^l} \) and to \( \frac{1}{2^{l+1}} \) in the \((l + 1)\)th level. We now analyze the size of leaf buckets with an overflow probability of \( 2^{-\lambda} \). We consider the general case where we add \( \eta < N \) leaf nodes to the ORAM.

**Lemma 4.2.2.** Let \( B_i \) denote the random variable describing the size of the \( i \)th leaf node, \( 1 \leq i \leq N + \eta \). For the dynamic strategy and a bucket of size \( B_i \), the overflow probability is given by

\[
\Pr(\exists i \in [N + \eta]: B_i \geq k) \leq \frac{2 \cdot N}{k + 1} \cdot \left( \frac{2 \cdot e \cdot \alpha}{k} \right)^k.
\]

**Proof.** After adding \( \eta \) leaf nodes to the structure, the ORAM contains \( N + \eta \) leaves. The probability that at least one leaf node has size larger than \( k \) is
\[
\Pr(\exists i \in [N + \eta] : B_i \geq k) = \Pr(\bigcup_{i=1}^{N+\eta} B_i \geq k) \leq \sum_{i=1}^{2\eta} \Pr(B_i \geq k) + \sum_{i=2\eta+1}^{N+\eta} \Pr(B_i \geq k) \quad (4.6)
\]

Note that the leaf nodes ranging from 1 to \(2 \cdot \eta\) are in the \((l + 1)\)th level with an assignment probability equal to \(\frac{1}{2^N}\) while leaves ranging from \(2 \cdot \eta + 1\) to \(N + \eta\) belong to the upper level and have an assignment probability equal to \(\frac{1}{N}\). Thus,

for \(1 \leq i \leq 2 \cdot \eta\) : \(\Pr(B_i \geq k) \leq \left(\frac{\alpha \cdot (N + \eta)}{k}\right) \cdot \left(\frac{1}{2 \cdot N}\right)^k\)

for \(2 \cdot \eta + 1 \leq i \leq N + \eta\) : \(\Pr(B_i \geq k) \leq \left(\frac{\alpha \cdot (N + \eta)}{k}\right) \cdot \left(\frac{1}{N}\right)^k\).

Note that \(\alpha \cdot (N + \eta)\) is the current number of elements in the ORAM. We plug both inequalities into Eq. 4.6 and get

\[
\Pr(\exists i \in [N + \eta] : B_i \geq k) \leq 2 \cdot \eta \cdot \left(\frac{\alpha \cdot (N + \eta)}{k}\right) \cdot \left(\frac{1}{2 \cdot N}\right)^k + (N - \eta) \cdot \left(\frac{\alpha \cdot (N + \eta)}{k}\right) \cdot \left(\frac{1}{N}\right)^k \\
\leq \left(\frac{2 \cdot \eta}{2^\eta} + N - \eta\right) \cdot (1 + \frac{\eta}{N})^k \cdot \left(\frac{\alpha}{k}\right)^k.
\]

The bound above depends on \(\eta\). Thus, we now compute the value of \(\eta < N\) maximizing the bound. This leads us to the function \(g(\eta) = \left(\frac{2 \cdot \eta}{2^\eta} + N - \eta\right) \cdot (1 + \frac{\eta}{N})^k \cdot \left(\frac{\alpha}{k}\right)^k\). Function \(g\) has a local maximum value for any \(\eta, 1 \leq \eta \leq N\) such that \(\eta_{max} = \frac{N}{A} \cdot \frac{k}{k+1}\) where \(A = 1 - \frac{1}{2^{1-\eta}}\). We replace \(\eta_{max}\) in \(g\) to get an upper bound for any \(\eta\) and \(k \geq 2\),

\[
\Pr(\exists i \in [N + n] : B_i \geq k) \leq g(\eta_{max}) \cdot \left(\frac{\alpha}{k}\right)^k \\
\leq N \cdot \frac{A + 1}{k+1} \cdot \frac{k}{A(k+1)}^k \cdot \left(\frac{\alpha}{k}\right)^k \\
\leq \frac{2 \cdot N}{k+1} \cdot \left(\frac{2 \cdot \epsilon \cdot \alpha}{k}\right)^k.
\]

As \(k \geq 2\), we conclude with \(\left(\frac{k(A+1)}{A(k+1)}\right)^k \leq 2^k\) and \(\frac{A+1}{k+1} \leq \frac{2}{k+1}\).

So, the overflow probability decreases exponentially when increasing bucket size \(k\). Note that, in the proof, we have maximized the overflow probability independently of the number of nodes.
added (which is a function of \( \eta \)). In practice, \( k \) could be smaller for some intervals of insertions, but we have chosen a maximal value to avoid issues related to changing the leaves’ size during insertions.

### 4.2.5 Comparison of Strategies

We present a comparison between our three strategies in terms of storage complexity (Figure 4.2) and communication complexity per access (Figure 4.1). We perform our comparison on a block level, thereby remaining independent of the actual block size.

**Communication complexity:** the lazy strategy offers best communication complexity. This is due to shorter paths, a result of flatter trees – compared to the naïve solution. Also, compared to the dynamic strategy, the leaf buckets have smaller size. For a number of elements \( N = 2^{30} \) and \( 2^{-64} \) overflow probability, the interior node size equals 36 which is appropriate for all three strategies. The difference consists on the size of the leaf buckets as well as the height of the resulting tree. The bucket size for the naïve, lazy and dynamic strategy respectively equals 28, 85 and 130 blocks. The tree’s height for the naïve solution equals 30 while for the lazy and dynamic solution the tree height is 26 since \( \alpha \approx 2^4 \). In Figure 4.1, for an eviction rate equals 4, the entire communication complexity (upload/download) on the main ORAM respectively equals 26928, 24210 and 25020 blocks for the naïve, lazy and dynamic solution. Note that per access, we save around 7% in communication cost. Recall that our main purpose is to reduce the storage overhead while maintaining the same communication complexity. However, our results show that storage optimization has a direct consequence on reducing the communication complexity as well.

**Storage complexity:** there is no “clear winner”. Depending on the client’s usage strategy, the dynamic (3rd) strategy can be considered best, as it provides more intuitive and fine grained control over storage size. However, if the insertion of elements follows a well defined pattern where the client is always expanding their capacity by a factor of \( \alpha \), the 2nd strategy will result in cheaper cost. The cost reduction is significant, around 87% fewer blocks compared to the naïve solutions.

Independent of the blocks size, this represents 87% of storage cost savings. Consider the following example: we fix the block size to 4096 Byte and the number of elements to \( N = 2^{30} \), resulting in a dataset size equal to 4 TByte. Based on Amazon S3 pricing [17] where the price is equal to 0.029 USD per GByte per month, the client has to store, for the naïve solution, \( \sim 2.8.10^{14} \approx 262 \) TByte, implying \( \sim 7600 \) (USD) per month. With the lazy solution, the client has to store only \( \sim 31 \) TBytes, which is only 900 (USD) per month (almost 10 times cheaper than the naïve solution).
In general, both the lazy and dynamic strategies outperform the naïve one in terms of communication and storage complexities.

4.2.6 Position Map

To maintain constant client memory, it is important to recursively store the mapping between tags and elements in a position map on the server. This position map is stored in a logarithm number of ORAMs with a number of leaves increasing exponentially from one ORAM to the other. With a position map factor $\tau$, $N = \tau^l$, the position map is composed of $l-1$ small ORAMs where ORAM$_i$ has a number of leaves equal to $\tau^i$, $1 \leq i \leq l - 1$.

Surprisingly, resizing the position map is trivial, e.g., following one of the two subsequent strategies: (1) use the same strategy of resizing (adding/pruning) that we apply on ORAM$_{l-1}$, or (2) create a new level of recursion in the case of adding, or deleting the last level of recursion in the case of pruning. Assume $N$ elements; each element is associated to a leaf tag that has size $\log N$ bits. We describe each solution for the case of the naïve adding strategy.

(1) When we add a new line to the main ORAM (ORAM$_l$), we have $2 \cdot N$ leaves instead of $N$ leaves. Similarly, we increase the size of the last ORAM of the position map (ORAM$_{l-1}$) to have a new level of leaves. The only issue with this solution is that we should increase the block size. Instead of having $O(\tau \cdot \log N)$ bits, it will have now $O(\tau^2 \cdot \log N)$ bits. Every time an element is accessed, the corresponding block is modified to have the new size. Note that when we add a new level of leaves, we can always access all elements of the ORAM using the previous mapping. For this, we just append at the end of the tag fetched an additional bit 0 or 1 to access a random child (to stay oblivious and access the entire path). After accessing any “old” elements (old denotes elements with a previous mapping), the mapping is updated to have $\log N + 1$ bits instead of $\log N$. 
The lazy solution is straightforward and based on creating a new level of recursion when a new level of leaves is created. Note that blocks in this level will have $O(\tau \cdot \log N + 1)$ bits instead $O(\tau \cdot \log N)$. To access an “old” element, we use the same method described above.

### 4.3 Pruning

Assume an ORAM storing $N$ elements. Now, the client deletes $\eta$ elements from the ORAM. Consequently, the naïve ORAM construction now contains $N - \eta$ elements, but still has $N$ leaves. Consequently, the client tries saving unnecessary storage costs and frees a number of nodes from the ORAM. Similar to adding element to the ORAM tree, we tackle pruning by presenting two different strategies. The first one, a lazy pruning, prunes the entire set of leaves of the lowest level $l$ and merges content with level $l - 1$. Our second strategy consists of a dynamic pruning that deletes two leaf nodes for a specific number of elements removed from the ORAM. Again, we will analyze overflow probabilities induced by such pruning as well as complexities.

#### 4.3.1 Lazy pruning

In Section 4.2.3, we have demonstrated that leaves can store significantly more elements while only slightly increasing their size. We will use this observation to construct a new algorithm for lazy pruning. Assume that the leaf level contains $N$ leaves for $\alpha \cdot N$ elements stored. Let $\eta$ denote the number of elements deleted by the client. For sake of simplicity, assume that, at the beginning, we have $\eta = 0$ and $N$ leaf nodes. Our pruning technique is similar to the “lazy” insertion described previously. Whenever $\alpha \cdot \frac{N}{2} < \eta \leq \alpha \cdot N$, we keep the same number of leaves. Within this interval, the client can add or delete elements without applying any change to the structure, as long as the number of elements remains within the defined interval. If the number of deletion equals $\alpha \cdot \frac{N}{2}$, the client proceeds to remove an entire level of leaf nodes. The client proceeds to read every leaf node, along with its sibling, and merges them with their parent node. While this appears to be straightforward, an oblivious merging of siblings into their parent is more complex under our constant-client memory constraint. We will discuss this in detail below.

The major problem of this technique is its unfortunate behavior in case of a pattern oscillating around the pruning value. For example, if the client deletes $\alpha \cdot \frac{N}{2}$ elements, prunes the entire level, then adds a new element back. In this case the ORAM structure has more than $\alpha \cdot \frac{N}{2}$ elements in $\frac{N}{2}$ leaves, so the client has to again double the number of leaves. This results in high communication costs.
4.3.2 Dynamic pruning

Given that pruning an entire level at once is very inefficient, we now investigate how pruning can be done in a more gradual way. For every $\alpha$ elements we delete, we will prune two children and merge their contents into their parent node. The pruning will fail if the number of elements in both children and parent is more than $k$. This can only occur if there are more than $k$ elements associated (tagged) to these children. The following lemma states the upper bound of the overflow probability for the parent node after a merging. Recall that we begin with a full binary tree of $N$ leaves and $\alpha \cdot N$ elements. Assume that we have already deleted $\alpha(\eta - 1)$ elements, and we want to delete an additional $\alpha$ elements.

Lemma 4.3.1. Let $P_\eta$ denote the random variable of the size of the $\eta^{th}$ parent node. For dynamic pruning, the probability that pruning will fail equals

$$\Pr(P_\eta > k) \leq \left(\frac{2e \cdot \alpha}{k}\right)^k$$

Proof. The pruning will fail iff there are more than a total of $k$ elements in the parent and the children. Any element in these three buckets must be tagged for either the left or the right child. In order to compute the overflow probability of the parent, we compute the probability that more than $k$ elements are tagged to both children.

$$\Pr(P_\eta > k) = \left(\frac{\alpha \cdot (N - \eta)}{k}\right) \cdot \left(\frac{2}{N}\right)^k$$

$$\leq \left(\frac{e \cdot \alpha \cdot (N - \eta)}{k}\right) \cdot \left(\frac{2}{N}\right)^k$$

$$\leq (1 - \frac{\eta}{N})^k \cdot \left(\frac{2e \cdot \alpha}{k}\right)^k$$

$$\leq \left(\frac{2e \cdot \alpha}{k}\right)^k$$

In conclusion, the probability decreases exponentially with bucket size $k$. The upper bound is independent of the number of pruned nodes $\eta$. In practice, the bounds are tighter, especially for larger values of $\eta$.

4.3.2.1 Complexity of oblivious merging

The cost of dynamic pruning boils down to the cost of obliviously merging three buckets of size $k$. We can achieve this with $O(k)$ communication and constant memory complexity. First, note that we do not have to merge all three buckets at once. All that is required is an algorithm which
**Input:** Configuration of buckets $A$ and $B$

**Output:** A permutation which randomly “lines up” bucket $B$ to bucket $A$

// Slots in $A$ and $B$ start either empty or full; mark slots in $A$ as “assigned” if block from $B$ is assigned in $\pi$

$x \leftarrow$ number of empty slots in $A$;

$y \leftarrow$ number of full slots in $B$;

$d \leftarrow x - y$;

for $i$ from 1 to $k$ do

  if $B[i]$ is full then
    $z \leftarrow$ all empty slots in $A$;
  else
    if $d > 0$ then
      $z \leftarrow$ all non-assigned slots in $A$;
      $d \leftarrow d - 1$;
    else
      $z \leftarrow$ all full slots in $A$;
    end
  end

$\pi[i] \leftarrow z$;

$A[z] \leftarrow$ assigned;

end

return $\pi$;

**Algorithm 6:** GeneratePermutation($A$, $B$)

Obliviously merges two buckets. We can then apply it to successively merge three buckets into one. Since the adversary already knows that the two buckets being merged have no more than $k$ elements in them (as shown above), the idea will be to retrieve the elements from each bucket in a more efficient way that takes advantage of this property.

In Algorithm 6, the client randomly permutes the order of the elements in one bucket, subject to the constraint that, for all indices, at most one of the elements between both buckets is real. That is, the permutation “lines up” the two buckets so that they can be merged efficiently. Special care must be given to generate this permutation using only constant memory. The client makes use of “configuration maps” which simply indicate, for every slot in a bucket, whether that slot is currently full or empty. These maps can be stored encrypted on the server and take up $O(1)$ space each in terms of blocks (because the buckets contain $O(\log N)$ elements and a single block is at least $\log N$ bits [5, 7]). Then, the client iterates through the slots in one bucket, randomly pairing them with compatible slots in the other (i.e., a full slot cannot be lined up with another full slot). An additional twist is that an empty slot can be lined up with either a full or empty slot in the other bucket, but not at the expense of “using up” an empty slot that might be needed later since we cannot match full with full. Therefore, we have to also keep a counter of the difference between empty slots in the target bucket and full slots in the source bucket.

As seen in Figure 4.3, once the client generates the permutation, they can retrieves the elements pairwise from both buckets (i.e., slot $i$ from one bucket and the slot which is mapped to $i$ via the permutation from the other bucket), writing back the single real one to the merged bucket.
It remains to show that this permutation does not reveal any information to the adversary. If it was a completely random permutation, it would certainly contain no information. However, we are choosing from a reduced set: all permutations which cause the bucket to “line up” with its sibling.

Fortunately, we can formally prove that our permutation does not reveal any information beyond what the adversary already knows. This is because there are no permutations which are inherently “special” and are more likely to occur, over all possible initial configurations of the bucket. For every permutation and load of a bucket, there are an equal number of bucket configurations (i.e., which slots contain real elements and which do not) for which that permutation is valid.

To make this approach work, we need to slightly modify the behavior of the bucket ORAMs. Previously, when a new element was added to a bucket, it did not matter which slot it went into that bucket. It was possible, for instance, that all the real elements would be kept at the top of the bucket and, when adding a new one, the client would simply insert that element into the first empty slot that it could find. However, to use this permutation method we require that the buckets be in a random “configuration” in terms of which slots are empty and which are filled. Therefore, when inserting an element, the client should choose randomly amongst the free slots. Again, this is possible with constant client memory using our configuration maps. With this behavior, applying the above logic leads to the conclusion that the adversary learns nothing about the load of the bucket from seeing the permutation.

### 4.3.3 Proof: Oblivious Permute-and-Merge

**Lemma 4.3.2.** Given two buckets with maximum size $k$ and load $m$ and $n$ respectively, over the random configurations of those buckets, Algorithm 6 will output a uniformly random permutation which is independent of $m$ and $n$. 

![Figure 4.3: Illustration of permute-and-merge process. Bucket (2) is permuted and then merged with bucket (1) to create a new, combined bucket (3).](image-url)
Proof. We can determine the probability of a particular permutation $\pi$ being chosen, given $m$ and $n$, with a counting argument. It will be equal to

$$\frac{\text{# of configurations for which } \pi \text{ is a valid permutation}}{\text{total # of configurations} \times \text{# of valid permutations for a given configuration}}$$

The number of configurations for which $\pi$ is a valid permutation depends on $m$ and $n$, but not on $\pi$ itself. This can be seen if you consider that applying the permutation to a fixed configuration of the bucket simply creates another, equally likely configuration. The number of configurations for the sibling bucket that will “match” with that bucket are exactly the same no matter what the actual configuration of the first bucket is. This, combined with the fact that the probabilities must sum to one, tells us immediately that every permutation is equally likely. However, we can continue and express the total quantity for our first expression as

$$\binom{k}{m} \binom{k - m}{n}$$

This can be thought of as choosing the $m$ full slots for one bucket freely and then choosing the $n$ full slots in the second bucket to line up with the free slots in the already chosen first bucket. The number of valid permutations per configuration can equally be determined via a counting argument as

$$\binom{k - m}{n} \cdot (k - n)! \cdot n!$$

That is, choosing free slots for the $n$ elements in the second bucket and then all permutations of those elements times the permutations of the free blocks. That gives us a final expression for the probability of choosing permutation $\pi$ of

$$\frac{\binom{k}{m} \binom{k - m}{n}}{\binom{k}{m} \binom{k}{n} \cdot (k - n)! \cdot n!} \quad \text{(4.7)}$$

With some algebraic computations, we can show that the Eq. 4.7 can be simplified to $\frac{1}{k!}$. That is, this shows that the number of permutations, for any random distribution of load in a bucket, is independent of the current load. Again, since this does not depend on $\pi$ (but only on the size of the bucket), every permutation must be equally likely over the random configurations of the buckets.

\[\square\]

Corollary 4.3.1. A permutation $\pi$ chosen by Algorithm 6 gives no information about the load of the buckets being merged.
Proof. By our above lemma, independent of the load each permutation is chosen uniformly over the configurations of the two buckets. Therefore the permutation cannot reveal any information about the load.

4.3.4 Privacy analysis

Theorem 4.3.1. Resizable ORAM is a secure ORAM following Definition 4.1.1, if every node is a secure trivial ORAM.

Proof. Given that ORAM buckets are secure trivial ORAMs, we have to show that two access patterns induced by $\vec{y}$ and $\vec{z}$ of the same length are indistinguishable. Compared to traditional ORAM, resizable ORAM includes two new operations, Alloc and Free. Note that those operations should be in the same positions for both sequences, otherwise, distinguishing between the access pattern will be straightforward. Furthermore, we have already shown that, for increasing the size of the ORAM, Alloc for the 2nd and 3rd strategies will not induce any leakage. Also, lazy or dynamic pruning strategies will not leak any information about the load of the buckets. That is, the Free operation is oblivious. So, these additional operations do not leak any other information besides the actual number of elements (or a window that bounds the current number of elements for strategies 1 and 2). Also, the access patterns induced by other operations in both sequences $\vec{y}$ and $\vec{z}$ are indistinguishable (see the proof by Shi et al. [5]). We can conclude that resizable ORAM is a secure ORAM following Definition 4.1.1.

4.4 Conclusion

We are the first to show how to dynamically resize constant-client memory tree-based Oblivious RAM. This allows for use cases where clients do not know in advance exactly how much storage they will need and/or wishes to scale their storage needs efficiently and cheaply. We have shown that the naïve solution of adding leaf nodes induces a significant, unnecessary overhead. Instead, more advanced strategies, lazy insertion and dynamic insertion, can save dramatically on communication and storage cost compared to the naïve solution, although neither strategy is clearly superior to the other. Furthermore, we have demonstrate that the size of a tree-based ORAM can be decreased efficiently using an oblivious pruning technique. We have rigorously analyzed the overflow probability for each technique and presented a tight analysis of both interior and leaf node sizes.
Chapter 5

Write-only Oblivious RAM and Applications to Hidden Volume Encryption

So far we have worked at addressing issues preventing the practical use of Oblivious RAM, one of the chief obstacles being the large communication overhead. ORAM is a very general tool which provides an extremely high level of security. For some applications, it may in fact provide more security than is necessary. In this chapter we will discuss ORAM under a relaxed security model we call write-only ORAM, where the adversary sees only the writes performed on the storage device and not the reads. We will identify some specific use cases where such a model is appropriate and show that with this relaxed level of security significant efficiency gains can be achieved.

In this chapter, we will also motivate the importance of a write-only setting by exploring disk encryption as an application for these specialized ORAMs. Disk encryption, on the whole, is an important security technology that is increasingly being used by individuals and businesses alike. All major operating systems now support basic encrypted volumes natively, and both corporations and governments are increasingly mandating [43] the use of full disk encryption. Additionally, there are open source software products, most prominently TrueCrypt [44], that provide more advanced solutions.

One of the advanced features that TrueCrypt offers is “hidden volume” encryption. Instead of a single encrypted volume, a user may choose to have two encrypted volumes. These volumes are encrypted with different keys (derived from passwords), and the user has the ability to plausibly deny the existence of the second volume. An adversary, knowing only the password to the first volume, cannot tell for sure whether there exists a second volume, let alone what its contents may
be. Given the widespread use of encrypted disks, this is a very useful feature. If an adversary takes possession of an encrypted disk, they know that there is at least some data on that disk. They can then coerce the user to reveal their password used to encrypt the disk. With a hidden volume, the user can reveal the password to the first volume while withholding the password for the second. The adversary will not know whether the second volume exists, and therefore cannot be sure if there even is a second password for the user to reveal.

TrueCrypt accomplishes this by storing the second, “hidden” volume inside the free space of the first, “main” volume. Since the semantics of TrueCrypt guarantee that all free space in the encrypted volume will be filled with random data, and the encryption used is presumed to be indistinguishable from random, an adversary cannot tell if the blocks marked “free” in the main volume are actually free or if they contain encrypted data that is part of a hidden volume.

However, as already noticed by Czeskis et al. [45], TrueCrypt’s approach has a significant flaw: if the adversary has the ability to take multiple “snapshots” of the hard disk at different times, they can determine with high probability whether a hidden volume exists. Since disk encryption is specifically designed to protect against scenarios where the user loses control of their device, this is a major drawback. It is unrealistic to expect that an adversary has access to the machine and associated hard disk only one time is unrealistic. For example, it is common for users to travel with their devices and lose direct possession on multiply occasions, either through carelessness or necessity.

Thus, a system whereby a user can plausibly deny the existence of a hidden volume needs to guard against such situations. The reason that TrueCrypt does not maintain security against multiple snapshots is that it makes no attempt to hide the pattern of accesses that one makes to the disk. That, combined with the fact that the hidden volume is stored separately from the main volume (in the free blocks of the main volume), gives the adversary a large advantage. An adversary can compare separate snapshots and see if a large number of “free” blocks have changed values. Changes indicate that they are actually encrypted blocks that are part of a hidden volume, since they would otherwise not have a reason to change spontaneously.

These weaknesses lead us to our first observation: a system that is secure against multiple snapshots must make some attempt to hide the user’s access pattern. This is where Oblivious RAM comes in. We will show how ORAM can be used to create exactly such a solution. Yet, a straightforward ORAM application comes with a significant read/write overhead. We will introduce a more efficient ORAM under our write-only model, which provides sufficient security for our purposes and has a significantly lower overhead than related work. Based upon this write-only ORAM, we finally present HIVE, a new scheme for Hidden Volume Encryption.

The technical highlights of this chapter are:
The first formal treatment for hidden volume encryption and security, including multiple adversarial models. In addition, we show that several intuitive notions of security against a strong adversary are impossible to achieve.

HIVE, a new solution which provides hidden volume encryption and provably achieves our notions of security using write-only ORAM.

A novel write-only ORAM construction which achieves optimal (constant) communication complexity.

An implementation of HIVE in the Linux kernel that is practical. Our implementation realizes hidden volumes as regular Linux block devices. We evaluate our implementation, and our benchmarks show that HIVE is both efficient and usable. The source code is available for download [46].

Plausibly Deniable Encryption: The hidden volume functionality offered by HIVE, TrueCrypt, and others [47–49] is also called plausibly deniable encryption. However, here we refrain from using this term so as not to confuse it with the slightly different ideas of deniable encryption by Canetti et al. [50]. Instead, we use hidden volume encryption.

5.1 Robust Hidden Volumes

We start by introducing the system and adversary model for hidden volume encryption. We envision a typical scenario with a user \( \mathcal{U} \) having read/write access to a block storage device, e.g., a hard disk, USB stick or a network block device. User \( \mathcal{U} \) is running a special software for hidden volume encryption. This software gives access to a sequence of independent volumes \( V_i \) that are mapped to the underlying storage device. To get access to these volumes, \( \mathcal{U} \) knows a sequence of passwords \( \mathcal{P} \). We stress that each \( P_i \in \mathcal{P} \) gives full access to hidden \( V_i \), i.e., encryption keys used in \( V_i \) are derived from passwords \( P_i \), and \( \mathcal{U} \) chooses the \( P_i \) carefully. We assume that each \( P_i \) is chosen securely and can be used to derive a key with at least \( s \) bits of entropy, where \( s \) is a security parameter [51].

5.1.1 Model

One of our main contributions is the first formalization for hidden volume encryption. A hidden volume encryption scheme \( \Sigma \) provides an interface to access \( \max(V) \) volumes. We generalize to \( \max(V) \) volumes instead of two for increased flexibility. Each volume \( V_i \) has a password \( P_i \) associated with it and holds \( n_i \) blocks of data, each of size \( B \). The size of the entire disk is \( N \) blocks. For simplicity, both volume blocks and hard disk blocks have the same size \( B \). Usually, \( B = 512 \) Byte, but \( B \) can be varied up to 4096 Byte as discussed later. We do not try to hide
the value of $max$, but rather allow the user, $U$, to choose some number of volumes $\ell \leq max$ which he will be actively using. It is this choice $\ell$ that will be hidden. Thus a hidden volume encryption scheme $\Sigma$ works in such a way that an adversary, seeing changes to the blocks of a hard disk and knowing one or more passwords, still has some uncertainty about $\ell$. We also assume that the user has $t$ blocks of RAM which are not visible to the adversary. To avoid the trivial solution of storing everything out of sight of the adversary, we constrict the size of the RAM to be much smaller than the size of the disk (i.e., logarithmic in $N$).

Typically, a hidden volume encryption scheme $\Sigma$ is embedded in an operating system. There, it provides the functionality of a block device driver, yet it resides on top of an underlying (hardware) block device which we now call the disk for simplicity.

**Definition 5.1** (Hidden Volume Encryption $\Sigma$). Let $s$ denote the security parameter, $t$ denote the number of available RAM blocks, and $P = (<P_1, \ldots, P_\ell>)$ denote the sequence of user passwords. A hidden volume encryption (HVE) scheme $\Sigma$ comprises the following algorithms.

- **HVESetup($s, t, P, B, <n_1, \ldots, n_\ell>$)**: Using parameters $s, t$, the sequence of passwords $P$, block size $B$, and the size of each volume $n_i$, this algorithm generates volumes $<V_1, \ldots, V_i, V_{i+1}, \ldots, V_{max}>$.

- **HVEWrite($b, d, i, P$)**: Using passwords $P$, if $i \leq \ell$, then this algorithm stores data $d$ at block index $b$ in volume $V_i$, where $V_i$ was output by HVESetup($s, P$).

- **HVERead($b, i, P$)**: Using passwords $P$, if $i \leq \ell$, then this algorithm returns data from the block indexed by $b$ in volume $V_i$, where $V_i$ was output by HVESetup($s, P$).

To allow a scheme $\Sigma$ to read from and to write to the disk, we assume availability of regular read and write system calls. In the rest of this chapter, we will use DiskRead($\beta$) to denote a read from block index $\beta$ of the disk and DiskWrite($\beta, d$) to denote a write of data $d$ to block $\beta$. Again for simplicity, we assume that a scheme $\Sigma$ creates, for each volume, a new virtual block device within the operating system, which can be formatted with a file system and used just like a regular device. Informally speaking, a scheme $\Sigma$ has to 1) translate OS reads and writes to one of the block devices (volumes) into calls of HVERead and HVEWrite, and 2) apply its logic to finally use DiskRead($\beta$) and DiskWrite($\beta$). Note that $b$ denotes the index of a virtual block in one of the volumes, while $\beta$ denotes the index of a physical block of the disk.

### 5.1.2 Security Definitions

We now formalize security for hidden volume encryption. To start, we define an **access** $o = (op, b, V, d)$. If $op = write$, then this access is a write to block $b$ with value $d$ in volume $V$, and if $op = read$, then this access is a read of block $b$ in volume $V$ which returns data value $d$. We
call the sequence of accesses $O = < o_1, \ldots, o_n >$ the access pattern. We also allow $o = \perp$, which is a “null” operation that is simply ignored. The function Execute is used to signify the HVE scheme performing the requested accesses on the disk.

Our formalization of different security levels uses game-based definitions. All games $\Gamma$ will be played between an adversary $A$ and a challenger $C$ running a scheme $\Sigma$. All games $\Gamma$ will adhere to the following generic game. We present the specific differences between the games and corresponding levels of security after the generic game.

In our games, an adversary $A$ is allowed to repeatedly retrieve snapshots of the disk. We define a snapshot as the entire contents of the disk (i.e., addresses and current values of every block), and a snapshot is meant to represent a dump or capture of the hard drive from a machine.

The generic game $\Gamma_{A,\Sigma}^{\text{generic}}(s)$, cf. Fig. 5.1, is defined as:

1. Adversary $A$ chooses $\ell \leq \text{max}$ and sends it to $C$.
2. Using security parameter $s$, $C$ chooses $\ell$ passwords and a random bit $b$. $C$ initializes two different hidden volume encryption schemes: $\Sigma_0$ (with $\ell$ passwords) and $\Sigma_1$ (with $\ell - 1$ passwords). Finally, $C$ sends passwords $< P_1, \ldots, P_{\ell-1} >$ and an initial snapshot $D_0$ of the disk to $A$.
3. $A$ chooses two accesses and sends them to $C$, along with a bit $d$ that specifies whether $A$ would like a snapshot of the disk after execution. Both the access patterns and bit $d$ will adhere to specific restrictions that we detail below.
4. Following scheme $\Sigma_b$, $C$ “executes” one of the two accesses, depending on bit $b$.
5. If $d = 1$, $C$ sends a snapshot $D_i$ of the disk to $A$.
6. Repeat steps 3 to 5 for $p = \text{poly}(s)$ times (rounds $i$). $O_{i,j}$ denotes access $j \in \{0, 1\}$ in round $i$, $1 \leq i \leq \text{poly}(s)$.
7. $A$ outputs bit $b'$. The outcome of $\Gamma_{A,\Sigma}^{\text{generic}}(s)$ is 1, iff $b' = b$.

Figure ?? illustrates this process, with labels corresponding to each of the above steps in the procedure.

From this general game, we develop two axes along which we can define varying levels of security: 1) the frequency with which $A$ can access snapshots of the disk (regulating when and how often $d_i$ can be 1 in our game above), and 2) the restriction applied to the two access patterns $O_0, O_1$ that $A$ chooses and submits to $C$. We denote by $\Gamma_{A,\Sigma}^{x,y}(s)$ a game that follows generic game $\Gamma_{A,\Sigma}^{\text{generic}}(s)$, but implies restrictions “$x$” (defined below) on the frequency of snapshots and restrictions “$y$” (defined below) on the access patterns.

**Definition 5.2** (Hidden Volume Encryption). A hidden volume encryption scheme $\Sigma$ is secure regarding restrictions $(x, y)$, iff for any probabilistic polynomial time adversary $A$ there exists a
In conclusion, the rationale behind this game-based definition is to prevent adversary $A$ from successfully guessing whether there exist $\ell - 1$ or $\ell$ volumes in use, even when $A$ is able to choose $\ell$. We allow $A$ to specify two patterns $O_0, O_1$, similar to classic indistinguishability proofs, which allows us to capture a stronger chosen plaintext adversary. The restrictions we place on these access patterns will lead to an intuitive understanding of the security our scheme provides.

5.1.2.1 Snapshot Frequency

An important aspect of the adversary’s ability is the frequency with which we allow them to take snapshots. TrueCrypt relies on the adversary have one-time access to the device, so we must certainly consider that in our models. In addition, we would like to consider adversaries which have stronger multi-snapshot capabilities.
### Table 5.1: Different Notions for Hidden Volume Encryption

<table>
<thead>
<tr>
<th></th>
<th>Restricted Hiding</th>
<th>Opportunistic Hiding</th>
<th>Plausible Hiding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary</td>
<td>Impossible</td>
<td>HIVE-B</td>
<td>HIVE</td>
</tr>
<tr>
<td>On-Event</td>
<td>Recryption</td>
<td>HIVE-B</td>
<td>HIVE</td>
</tr>
<tr>
<td>One-Time</td>
<td>TrueCrypt, HIVE</td>
<td>TrueCrypt, HIVE</td>
<td>TrueCrypt, HIVE</td>
</tr>
</tbody>
</table>

1. **Arbitrary**: \( \mathcal{A} \) can obtain snapshots after every operation to the disk (any number of \( d_i \) can be 1). This is the most powerful type of snapshotting capability, but may be stronger than necessary.

2. **On-Event**: \( \mathcal{A} \) can obtain snapshots after \( U \) has run an “unmount” operation. We represent this by having an additional function Unmount which is called after Execute, whenever \( d_i = 1 \), to generate the snapshot.

3. **One-Time**: \( \mathcal{A} \) can obtain only a single snapshot (\( d_i = 1 \) for only a single value of \( i \)).

Our justification for also considering an “on-event” adversary is that, in reality, some adversaries will not have the capability to take arbitrary snapshots of the disk while it is in use. More likely, the machine will be confiscated or compromised while \( U \) does not have the hidden volume mounted. Therefore, a model where the adversary can only take snapshots after an “unmount” is interesting. Conceptually, it could make the problem easier since the system does not need to be in a secure state at all times, only after the unmount is performed. Security operations could be delayed until this point, and need not be performed constantly.

#### 5.1.2.2 Access Pattern Choice

An important part of our formal security is allowing the adversary to choose some part of the access pattern. This represents the fact that, in reality, an adversary may have partial knowledge of \( U \)’s access pattern. Informally, allowing adversary \( \mathcal{A} \) to choose the access pattern will guarantee that no matter what \emph{a priori} knowledge \( \mathcal{A} \) might have, they still cannot learn the secret we are trying to protect (i.e., whether there is a volume still unrevealed). To formalize this, we allow \( \mathcal{A} \) to choose a value \( \ell \), between 1 and \( \text{max} \) which represents the number of volumes in use by \( U \). To maintain some uncertainty about exactly how many volumes are actually being used, we then allow \( \mathcal{A} \) to also choose two access patterns, one which includes accesses to volume \( \ell \) and one which does not.

We stress that restrictions on access pattern choices are mandatory to prevent “trivial” impossibility. For instance, in our model \( \mathcal{A} \) will get the passwords for all volumes up to \( V_{\ell-1} \). If the two patterns contain different writes to these volumes, then the resulting snapshots will allow \( \mathcal{A} \) to distinguish between them easily (i.e., he simply decrypts these volumes and checks which values
were written). This is to be expected, since the point of hidden volume encryption is to protect accesses to the secret volumes which may or may not exist, from the adversary’s perspective. Once a password is given up, we cannot hide the contents any longer.

Therefore, the first restriction which must exist for all of our definitions is that any access where both patterns write to a volume $V_1$ through $V_{\ell-1}$ must contain identical writes in both patterns chosen by adversary $A$.

We start by presenting the most straightforward and powerful type of adversary which may choose all the accesses in any way they like. We then show that this notion of security is too strong and in fact impossible to achieve. Subsequently, we restrict the choice ability that the adversary has in two different ways to produce meaningful adversaries with real-world analogues that allow for constructive solutions. The restrictions in each settings are presented as $A$ being able to choose an access pattern $O_0$ and then a second pattern $O_1$ that must constrain to some restrictions based on $O_0$.

**Restricted Hiding**: $\Gamma_{\mathcal{A},\Sigma}^{(\cdot)}_{\text{Restricted}}$ The first setting we consider is the most straightforward: in each round, we allow $A$ to choose any access pattern $O_0 = < o_{0,1}, \ldots, o_{1,n} >$. As stated before, if access $o_{0,j}$ is a write in volume $V_j$, $j \leq \ell - 1$, then access $o_{1,j}$ in pattern $O_1$ must be equal to $o_{0,j}$. If $o_{0,i}$ is in $V_\ell$, then $o_{1,i}$ must be $\bot$ (the null operation, simply ignored by $C$). The access pattern executed when $b = 1$ is then the same as when $b = 0$, with all accesses to $V_\ell$ ignored by $C$.

Since the difference between $O_0$ and $O_1$ is only the removal of all accesses to $V_\ell$, a Restricted Hiding scheme effectively prevents an adversary from distinguishing between the case where a user uses $\ell$ volumes and the case where a user only uses $\ell - 1$ volumes, with no restriction on the user’s access pattern. This means that a scheme with Restricted Hiding would be the ideal system, since it would protect any access pattern and an adversary would not be able to learn whether there are $\ell$ or $\ell - 1$ volumes in use. Unfortunately, this level of security is too strong and we can show that it is impossible to achieve:

**Lemma 5.3.** Let $n^*$ be the number of blocks of volume $V_\ell$. There is no scheme $\Sigma$ that is $\Gamma_{\mathcal{A},\Sigma}^{(\cdot)}_{\text{Arbitrary,Restricted}}$-secure and requires less than $n^*$ blocks of RAM.

*By counterexample.* Assume that a secure scheme $\Sigma$ exists. We can construct an adversary $A$ which breaks $\Sigma$. First, $A$ submits two access patterns of length $n^*$, $O_0$ and $O_1$, where $O_{0,i} = (\text{write}, i, \ell, r_i)$, $O_{1,i} = \bot$ and $r_i \overset{\$}{\leftarrow} \{0,1\}^B$. If $A$ observes changes to the disk, $A$ outputs 1. Otherwise, $A$ outputs 0. If $C$ executes $O_1$, nothing on the disk will change because $\bot$ operations are ignored. If $C$ executes $O_0$, then either the disk must change or $C$ must have at least $N^*$ blocks of RAM to hold all the writes from $O_0$, which contradicts the assumption that $\Sigma$ is a secure scheme with $O(\log N)$ blocks of memory. \hfill $\Box$
This essentially means that against an adversary with Arbitrary snapshotting capability we can do no better than storing all the hidden volumes in RAM, which is quite unrealistic.

Although none of the solutions here are directly targeted at On-Event security, we include it in our definitions because it is a distinctly separate adversarial model from Arbitrary and One-Time. It has at least one interesting property: even though it is impossible to achieve \( \Gamma^{A,\text{Restricted}}_{\text{Arbitrary}} \) security, it is quite simple (albeit inefficient) to obtain \( \Gamma^{\text{On-Event,Restricted}}_{A,\Sigma} \) security. If a TrueCrypt-like approach is taken except, upon unmount, every data block is reencrypted and every empty block is filled with a uniformly random string, we achieve this security. This is highly inefficient, but we point it out as an interesting observation on our various adversarial models and possible motivation for more efficient solutions in the future.

**Opportunistic Hiding:** \( \Gamma^{\text{Opportunistic}}_{A,\Sigma} \)

The second setting we consider, opportunistic hiding, is similar to restricted hiding, but with a slightly more specific access pattern: again, if \( o_{0,i} \) is in volume \( V_j, j \leq \ell - 1 \), then \( o_{1,i} \) must be equal to \( o_{0,i} \). Also, if \( o_{0,i} \) is in \( V_\ell \), then \( o_{1,i} \) will be \( \perp \). The additional restriction is that between snapshots, every write to a volume \( V_2 \) through \( V_\ell \) in \( O_0 \) must have a corresponding read to volume \( V_1 \) which occurs after it. More formally, the additional restriction says that for \( O_0 = (o_1, \ldots, o_n) \), there must exist a one-to-one mapping \( f : [1 \ldots n] \rightarrow [1 \ldots n] \) such that if \( o_{0,i} \) is a write to a volume higher than \( V_1 \), then \( f(i) > i \) and \( O_{0,f(i)} \) is a read from volume \( V_1 \).

What this effectively means is that the system will hide writes to volumes higher than \( V_1 \) by executing them simultaneously with reads to \( V_1 \). This reflects the idea that, if we make execution of reads and writes similar, there will be extra capacity during a read to simultaneously do a write. This constraint is reasonable because it is very reasonable that the “secret” volumes will be accessed much less frequently than the public volume, which contains operating system files, application files, etc.

This security definition also provides “complete” security to the user, in that an adversary should not be able to distinguish whether there are \( \ell - 1 \) or \( \ell \) volumes. In that way, it is similar to our first definition, however, in order for the user to receive this security, it requires that the access pattern \( O_0 \) must have fewer accesses to \( V_2 \) through \( V_\ell \) than \( V_1 \). In practice this means that the user must access volume \( V_1 \) more often than the other, more secret volumes.

**Plausible Hiding:** \( \Gamma^{\text{Plausible}}_{A,\Sigma} \)

Finally, we consider an even more restricted setting where writes to \( V_\ell \) can be plausibly denied as operations to the other volumes \( V_1, \ldots, V_{\ell-1} \). \( A \) may choose any access \( O_{0,i} \). Again, if \( O_{0,i} \) is a write in volume \( V_j, j \leq \ell - 1 \), then \( o_{1,i} \) must be equal to \( O_{0,i} \). Otherwise, we only require that neither pattern contains \( \perp \) operations (so their “true” lengths are equal).
The intuition of this security definition is that, if an access pattern contains writes to \( V \), there is always a plausible access pattern containing only accesses to \( V_1 \) through \( V_{\ell-1} \) (with writes to \( V \) replaced by reads to other volumes) which would have produced the same sequence of disk operations. Therefore an adversary can never be sure of the existence of \( V \). Additionally, all writes to \( V \) will be indistinguishable from each other. In contrast with the previous two definitions, we have restrictions on \( O_1 \) beyond that the accesses to lower volumes must be equal.

Finally, Table 5.1 summarizes our results. Our new scheme, HIVE, provides plausible hiding with arbitrary snapshots. A variant of HIVE, HIVE-B, provides opportunistic hiding with arbitrary snapshots (as well as weaker notions). TrueCrypt and other related work [44, 47–49, 52, 53] only provide security against one-time snapshots [45]. Additionally, as noted above, one can achieve on-event, restricted security with an expensive reencryption technique. Note that Arbitrary \( \Rightarrow \) On-Event \( \Rightarrow \) One-Time, and Restricted \( \Rightarrow \) Opportunistic.

We would like to stress that any hidden volumes scheme needs additional requirements to maintain security. For example, the OS and applications should not keep any trace of previously mounted hidden volumes, e.g., in a “recently opened documents” list. For a detailed discussion, we refer to Czeskis et al. [45] and [54].

### 5.2 Generic Hidden Volume Encryption

Current hidden volume encryption solution only protect against One-Time adversaries. We now present a first “generic” protocol, using Oblivious RAM as a building block, that offers stronger, more robust security against One-Time and Arbitrary adversaries.

The goal of our first, “generic” hidden volume encryption is two-fold: to hide the pattern of writes issued by the user and to give a plausible reason to access the disk when writing into hidden volumes. Broadly, we accomplish the first by using ORAM and the second by making a read to a volume cause a “dummy write”. This dummy will cover for a write to a hidden volume, and is in line with the idea that all operations should “look the same” regardless of which volume it is in, or even if it is a read or write.

To start, we use \( \max \) separate ORAMs (\( \text{ORAM}_{i \leq i \leq \max} \)), each holding the data for a single volume and encrypted with its own password, e.g., we simply run \( \text{ORAMSetup} \) for each of these ORAMs. The only requirement we have on our ORAM scheme is that it be efficiently simulatable, i.e., a simulator \( S \), without the password or any knowledge of the access pattern beyond its length can output a series of disk accesses which are indistinguishable from those output by an actual ORAM. Additionally, this simulator needs to be stateless, in that its output
Input: Security parameter $s$, RAM size $t$, passwords $\mathcal{P}$, block size $B$, volumes sizes $< n_1, \ldots, n_\ell >$

Output: Volumes $< V_1, \ldots, V_\ell, V_{\ell+1}, \ldots, V_{\max} >$

largestVolume := maximum($n_1, \ldots, n_\ell$);
for $i := 1$ to $\max$ do
  $V_i$ := ORAMSetup(largestVolume, $B, P_i, s$);
return $< V_1, \ldots, V_{\max} >$

Algorithm 7: Generic HVESetup($s, t, \mathcal{P}, B, < n_1, \ldots, n_\ell >$)

Input: Block $b$, data $d$, volume index $i$, passwords $\mathcal{P}$
for all the $j$ do
  if $j = i$ then
    ORAM_i.ORMWrite($b, d$);
  end
  else
    $r$ \inophile $\{1, \ldots, \text{sizeof}(\text{ORAM}_i)\}$;
    // Using password $P_j$ as key
dummy := ORAM_j.ORMRead($r$);
    ORAM_j.ORMWrite($r, \text{dummy}$);
end

Algorithm 8: Generic HVEWrite($b, d, i, \mathcal{P}$)

Input: Block index $b$, volume $i$, passwords $\mathcal{P}$
Output: Data $d$
$d$ := ORAM_i.ORMRead($b$);
for all the $i$ do
  $r$ \inophile $\{1, \ldots, \text{sizeof}(\text{ORAM}_i)\}$;
    // Using password $P_i$ as key
dummy := ORAM_i.ORMRead($r$);
    ORAM_i.ORMWrite($r, \text{dummy}$);
end
return $d$;

Algorithm 9: Generic HVERead($b, i, \mathcal{P}$)

for each operation cannot depend on anything other than the output from the previous operations. Fortunately, typical ORAM constructions meet this definition. So, for $j > \ell$, ORAM$_j$ is replaced by a simulator $S$. Therewith, an ORAM can execute a “dummy” operation containing no information, which, to all adversaries, looks identical to a real operation. This is necessary so that the ORAMs not actually in use by the user will not even have keys that could be revealed to the adversary. This gives the user deniability, since they can reasonably claim that no key exists for a certain volume.

When the system executes a write to volume $i$ (Algorithm 8), it executes that write on ORAM$_i$. For all $j \neq i$, it picks a random block in $V_j$ and writes its same data value back to ORAM$_j$, i.e., a “dummy” operation which changes no data. When the system executes a read (Algorithm 9), it reads from the respective ORAM and then does a dummy write for all ORAMs (or executes the simulator, for volumes greater than $\ell$). The idea is that, if an adversary does not have a key to volume $i$, they cannot tell whether we are reading from some volume (maybe $i$, but maybe not) or writing to volume $i$. For the internal encryption and decryption within ORAM$_i$, we use password $P_i$ as the key.
Theorem 5.4. If \((\text{ORAMSetup}, \text{ORAMRead}, \text{ORAMWrite})\) is a simulatable ORAM, then our generic hidden volume encryption is a \(\Gamma_{A,\Sigma}^{\text{Arbitrary,Plausible}}\)-secure hidden volume encryption.

Proof. Under Plausible Hiding security, the access patterns given by \(A\) will differ only when neither \(O_{0,i}\) nor \(O_{1,i}\) is a write to a volume less than \(\ell\). \(A\) cannot distinguish on operations of the access patterns that are identical, we only need to show that, for operations that differ, \(A\) cannot distinguish. So, if \(O_{0,i}\) and \(O_{1,i}\) differ, then they can each be either a read to any volume or a write to a volume \(j \geq \ell\). By the security definition of ORAM, a “dummy” write in a volume \(j \geq \ell\) is not distinguishable from an actual write with probability greater than \(1/2 + \epsilon(s)\). This implies that a read to one of these volumes cannot be distinguished from a write. We also have that for all \(i, j\), a read to \(V_i\) is indistinguishable from a read to \(V_j\). Therefore, since \(A\) cannot distinguish between the outputs with probability greater than \(1/2 + \epsilon(s)\), they cannot win the game with any non-negligible advantage.

5.2.1 Opportunistic Hiding Security

Opportunistic security gives more freedom to the user, since it does not require them to “pretend” that they did some reads that were actually writes in other volumes. Thus, it is relatively simple to achieve. Instead of writing a block immediately when the user wants to, if it is part of a volume \(V_i, i > 1\), we add it to a queue \(Q_i\). Every time the user does an operation on \(V_1\) (read or write), for all \(i > 1\) if \(Q_i\) is not empty, we write one block from \(Q_i\) to \(V_i\), instead of doing the dummy write. Reads to volumes other than \(V_1\) are trivial, i.e., we read the requested block, but do not change anything on the disk. Writing during such a reads is no longer necessary, because writes to all volumes higher than \(V_1\) are simultaneously hidden by accesses to \(V_1\).

Theorem 5.5. If \((\text{ORAMSetup}, \text{ORAMRead}, \text{ORAMWrite})\) is a simulatable ORAM, then this modified generic hidden volume encryption is \(\Gamma_{A,\Sigma}^{\text{Arbitrary,Opportunistic}}\)-secure.

Proof. \(O_0\) and \(O_1\) are only different when \(O_{0,i}\) is an operation in \(V_\ell\). If \(O_{0,i}\) is a read, it will be identical to a \(\bot\) operation in our modified generic scheme. Such a read does not trigger any write. If \(O_{0,i}\) is a write, it equally triggers no writes, as it just adds a block to the \(Q_\ell\), and again is indistinguishable. Therefore, we only need to show that a read to \(V_1\) which also writes from \(Q_\ell\) is indistinguishable from one which just causes a dummy operation in \(V_\ell\). This follows directly from the security of our ORAM, and so \(A\) cannot win the game with any non-negligible advantage.
5.3 Write-only ORAM

We have proven that our generic hidden volume encryption secure using ORAM as a building block. Note, however, that the snapshots $A$ gets in our security game do not include any information about block reads that occur to the disk. This reflects the idea that an adversary can see the impact of block writes, in the form of modified data, but they cannot see where or how often user $U$ reads. Typically, block reads do not leave any discernible trace on the disk. This means that the ORAMs we are using are actually more powerful than we need them to be. In fact, an ORAM which only hides writes to the disk is sufficient. We can define the security of such a write-only ORAM as follows:

**Definition 5.6 (Write-only ORAM Security).** Let sequence $\text{ExecW}(O)$ be the sequence of writes resulting from an ORAM executing access pattern $O$. A write-only ORAM (with algorithms $\text{ORAMSetup}$, $\text{ORAMRead}$, $\text{ORAMWrite}$) is secure iff, for any probabilistic polynomial time adversary $A$ and any two access patterns $O_0$ and $O_1$ that contain the same number of writes, there exists a function $\epsilon$ negligible in security parameter $s$, such that

$$|\Pr[A(\text{ExecW}(O_0)) = 1] - \Pr[A(\text{ExecW}(O_1)) = 1]| \leq \epsilon(s).$$

There has been limited work on write-only ORAMs until now. There are several schemes by Li and Datta [55], but they all have significant drawbacks. They are able to obtain an amortized write communication complexity of $O(B \cdot \log N)$, but only at the cost of reads being in $O(B \cdot n)$. As ORAM communication complexities directly relate to the hidden volume encryption overhead, this does not suit our needs. Applications might perform as many reads as writes, and reads would quickly become too expensive for increasing $n$. For efficient reads, Li and Datta [55] require either memory or communication complexity to be polynomial in $N$. We stress that their schemes only provide *amortized* complexity guarantees, with worst-case complexity being polynomial in $n$.

Since we target good read and write performance for our hidden volume encryption to be useful, we now present a new write-only ORAM which achieves worst-case constant communication complexity and only poly-logarithmic memory requirements. We will start with a very simple, inefficient construction and show how its shortcomings can be addressed one at a time until we have our final, efficient construction HIVE.

5.3.1 Basic Write-Only ORAM Construction

We now present our new ORAM that only supports write operations. Our ORAM makes use of a mapping data structure $\text{Map}$ that maps blocks from the ORAM to physical blocks (sectors)
on the disk. For now, we will consider it as an associative array such that \( \text{Map}[b] \) contains the physical address \( \beta \) on the disk where ORAM block \( b \) is currently located. Later, we will show how this map is structured and actually implemented, but for now assume that it is simply stored in RAM and can be efficiently accessed. Also assume that the hard disk has at least twice the size of the ORAM, i.e., \( N \geq 2 \cdot n \), so we have at least twice as much storage available as needed.

All mapping entries \( \text{Map}[b] \) are set to \( \bot \). For encryption and decryption, we employ any pair \( \text{Enc}, \text{Dec} \) of algorithms realizing IND-CPA encryption [56]. An IND-CPA encryption is an encryption that produces ciphertexts indistinguishable from random strings, e.g., AES in CBC or counter mode. The encryption makes use of secret key \( \kappa \), only known to the ORAM user \( U \).

**ORAMWrite**: For an ORAMWrite\((b, d)\) operation (see Algorithm 10), \( U \) picks a sequence \( S \) of \( k \) random, distinct hard disk block indices \( \beta_i \), where \( k \) is a security parameter. \( U \) then randomly picks one index \( \beta \in S \) that is “free”. Here, free means that there is no block in the ORAM that is mapped to disk block \( \beta \). User \( U \) writes \( \text{Enc}_\kappa(d) \) at position \( \beta \) to disk. Of the \( k - 1 \) remaining block indices, if their corresponding blocks contain encrypted data, \( U \) reencrypts their contents, and if the blocks are free \( U \) writes a random strings into them (“ReencryptOrRandomize”). Finally, \( U \) updates the mapping for \( b \). Since \( U \) picks the \( k \) block indices in \( S \) randomly and independently from \( b \), and the encryption produces ciphertexts indistinguishable from random, an adversary seeing these \( k \) blocks change cannot learn anything about \( b \) or \( d \).

**ORAMRead**: Our ORAM does not aim at protecting read operations. Thus, reads are trivial as shown in Algorithm 11.

**ORAMSetup**: For initialization, we require only that the map be initialized to an “empty” state, e.g. looking up the address of any block should return a \( \bot \) value indicating that the block has not been written in the system yet.

**Choice of \( k \)**: To guarantee that \( U \) always finds at least one free block in \( S \) to put data \( d \) into, \( U \) has to choose \( k \) sufficiently large. Since at least half the disk is empty, the probability that any randomly chosen block is free is at least \( 1/2 \). Let \( X \) be the random variable that, when selecting \( k \) blocks uniformly from all \( N \), describes the number of blocks among those \( k \) that are free. As \( N \) is typically large compared to \( k \), we approximate the hypergeometrically distributed \( X \) with a binomial distributed \( X \). \( \Pr[X \geq 1] = 1 - \Pr[X = 0] \approx 1 - \binom{k}{0} \cdot \left( \frac{n}{N} \right)^k = 1 - 2^{-k} \) for \( N = 2 \cdot n \). Therefore, if we set \( k \) equal to our security parameter \( s \), the probability of not finding at least one free block for any single write will be \( 2^{-s} \), which is negligibly small in \( s \). Setting \( N \) to twice \( n \) will “only” double storage requirements, but leads to high efficiency and practicality as we will show in Section 5.3.2.

**Free Blocks**: A remaining detail is how to determine which physical blocks \( \beta \) on the disk are actually “free”, so no block \( b \) of the ORAM maps to them. The challenge is to do this in an efficient way in order to keep a low complexity. A simple solution is a reverse mapping that tags
Input: Block index \( b \), data \( d \)
\( S := \{ \beta_i \} \), such that \( \beta_i \in \{ 1, \ldots, N \} \) and \( 1 \leq i \leq k \) such that \( \beta_i \neq \beta_v \) holds;
\( \beta \in S \), such that \( \beta \) is free;
\( \text{DiskWrite}(\beta, \text{Enc}_\kappa(d)) \);
\( \text{ReencryptOrRandomize}(S \setminus \beta) \);
\( \text{// Find out which block in the Map ORAM has the address we want} \)
\( M := \left\lfloor \frac{B}{\log N} \right\rfloor ; \)
\( \text{mapblock} := \text{Map.ORAMRead}(\left\lfloor \frac{b}{M} \right\rfloor) ; \)
\( \text{// Retrieve the address from the map block} \)
\( \text{mapblock}[b \mod M] := \beta ; \)
\( \text{Map.ORAMWrite}(\left\lfloor \frac{b}{M} \right\rfloor, \text{mapblock}) ; \)

Algorithm 10: ORAMWrite\((b, d)\)

Input: Block index \( b \)
\( M := \left\lfloor \frac{B}{\log N} \right\rfloor ; \)
\( \text{mapblock} := \text{Map.ORAMRead}(\left\lfloor \frac{b}{M} \right\rfloor) ; \)
\( \beta := \text{mapblock}[b \mod M] ; \)
\( \text{return Dec}_\kappa(\text{DiskRead}(\beta)) ; \)

Algorithm 11: ORAMRead\((b)\)

For each block \( \beta \) on the disk with the index for the ORAM block that maps to it. For an operation ORAMWrite\((b, d)\), what \( \mathcal{U} \) actually writes to disk block \( \beta \) is \( \text{Enc}_\kappa(d) || \text{Enc}_\kappa(b) \). When \( \mathcal{U} \) needs to determine if a disk block \( \beta \) is free, \( \mathcal{U} \) decrypts \( \beta \)'s content to restore ORAM address \( b \) and checks if \( \text{Map}[b] = \beta \). If that condition is not true, then \( \beta \) is an “old” version of ORAM block \( b \), so \( \beta \) is free, and it can be safely overwritten. Consequently, \( \mathcal{U} \) can therewith check if a block is free in constant time.

In practice, we cannot write the two ciphertexts \( \text{Enc}_\kappa(d) || \text{Enc}_\kappa(b) \) into a single block of size \( B \), as \(|d| = B \) already. Also, for each IND$^n$-CPA encryption, we need to store random coins such as an IV or counter. Consequently, we write the ciphertext of \( \text{Enc}_\kappa(d) \) into block \( \beta \), and we write \( \text{Enc}_\kappa(b) \) and the encryptions’ random coins into another metadata block \( \beta' \). One can imagine that, in addition to the \( N \) hard disk blocks for the write-only ORAM, we need additional \( N \) blocks for metadata. Each time an ORAM block \( \beta \) is written to disk, the corresponding metadata block \( \beta' \) is also updated. As the mapping between an ORAM block and its metadata block is fixed, this does not have any consequences for security. A straightforward optimization of this idea is to store multiple IVs and multiple \( \text{Enc}_\kappa(b) \) for multiple ORAM blocks in a single metadata block \( \beta' \), as typically \(|IV| + |\text{Enc}_\kappa(b)| < B \).

In conclusion, \( \mathcal{U} \) writes a block \( b \) into one of the \( k \) blocks from \( S \) chosen randomly and thus independently of \( b \), updating Map accordingly. Since \( k \) is a security parameter independent from \( n \), we only need to actually write a constant \( O(1) \) number of blocks to disk for each ORAM write operation. Still, our write-only ORAM has two drawbacks: first, it requires us to access a large number of blocks per operation (\( k \) is, for example, 64), and second, it requires \( O(n \cdot \log N) \) memory complexity to store the Map.
We now move on to optimizing our initial write-only ORAM to allow for smaller $k$ and for efficient storage of the map.

### 5.3.2 Stash-Optimized Write-Only ORAM

We have set $k$ to be equal to a security parameter $s$. This is to ensure that with probability $1 - 2^s$ $\mathcal{U}$ will choose a free block to write new data into for every write operation. It turns out that this is quite wasteful, because on average, $\mathcal{U}$ will find $s/2$ free blocks per operation – many more than are necessary. We only have to set $k$ so high to avoid a potential worst-case situation where $\mathcal{U}$ does not find any free blocks.

We now exploit the fact that typically there will be many more blocks free than just one. We set $k$ such that on average we will have one (or slightly more than one) free blocks, e.g., by setting $k$ to only 4. The intuition is that in that case, there will be many times where $\mathcal{U}$ does not find a free block, but there will even more times when $\mathcal{U}$ finds more than one free block. We can take advantage of this situation by having a small stash of pending blocks in memory. If $\mathcal{U}$ chooses $k$ blocks, and none of them are free, then $\mathcal{U}$ puts the block to be written at that time in the stash, to be written later. If $\mathcal{U}$ ends up with more than one free block during a write, then $\mathcal{U}$ additionally writes out some extra blocks from the stash. Note that this idea and its analysis is different from the one by Stefanov et al. [7], where a stash is used in case one of the bucket ORAMs is full.

The first challenge is to bound the size of the stash, such that we do not overburden user $\mathcal{U}$’s memory, but still have high probability of the stash not overflowing. To determine the stash’s size, we use a standard queueing argument. Our stash can be modeled as a $D/M/1$ queue with deterministic arrival rate $\gamma = 1$ and service times exponentially distributed with rate parameter $\mu = k/2$. As shown by Jansson [57], we can then express the steady state probability $P$ of having $i$ items in the stash at any time as $P = (1 - \delta) \cdot \delta^i$, where $\delta$ is the root of the equation $\delta = 2^{-\mu \gamma (1 - \delta)}$ with the smallest absolute value. If $\mu$ is larger than 1, then $\delta < 1$, and the steady state probability of having $i$ blocks in the stash will be $O(2^{-i})$. That means that the size of the stash will be $O(B \cdot s)$ with probability all but negligible in $s$.

This optimization allows us a huge savings in performance. We can go from $k = s$ to $k$ being a small constant, independent of the security parameter. For example, if $k = 3$, we can solve the equation to find $\delta = 0.41718$, and we can bound the probability of overflowing the stash at $2^{-64}$ using only a stash of size 50 blocks. With a block/sector size of 4096 Byte, the stash would consume only 200 KByte RAM.

**Security:** The actual writing that $\mathcal{U}$ performs to the $k$ blocks of $\mathcal{S}$ is hidden by the security of the IND$\$-$\$CPA encryption. Therefore, whether $\mathcal{U}$ writes one, two, or no blocks from the stash on
any given operation is not observable by the adversary. Consequently, the stash does not impact security at all.

5.3.3 Recursive map

As we have described our write-only ORAM thus far, we have good communication complexity, but at the cost of a large map that must be kept in memory. The total size of the map will be \( O(B \cdot n \cdot \log N) \), prohibitively large if we hope to store it in memory. However, we can use a standard technique [5, 39, 55] which is to recursively store the map in smaller and smaller ORAMs. If our block size \( B \) is at least \( \chi \cdot \log N \) for some constant \( \chi > 2 \), then we are guaranteed that if we (recursively) store our map in another ORAM, that ORAM in turn will have a map that is no greater than half the size of the original map. Therefore, after \( O(\log n) \) recursive ORAMs, we will have a map that is constant size and can be stored in memory.

This reduces the map to a much more comfortable size, but at the expense of increasing the communication complexity: now, to access the map, we have to, in turn, access \( O(\log n) \) recursive ORAMs. This will also slightly impact our stash analysis, since the ORAMs that hold the map do not have deterministic arrival rates, since their arrivals are in fact the result of service on the above queue. Fortunately, we can model them as \( M/M/1 \) queues with expected arrival rate equal to 1 which still results in exponentially distributed stationary probabilities [58], and we end up with the same \( O(\log n) \) bound on the size.

As seen in algorithms 11 and 10, we can then simply treat the map for each level as another ORAM and issue ORAMRead and ORAMWrite calls as necessary. Each level is unaware of how the next level structures itself, only that it provides an interface to read and write, and that it can do so securely. However, we cannot treat the map as a simple associative array now because the recursive ORAM will need to store more than one address per block in order to guarantee that we have only \( O(\log n) \) levels of recursion. Therefore, to read an entry from the map, we have to calculate \( N = \lfloor \frac{B}{\log n} \rfloor \), the number of entries we can fit per block, then figure out which block the entry we want will be in. For instance, if we want the address for block \( i \), we would get block \( \lfloor \frac{i}{N} \rfloor \) from the map, and read the \( i \mod N \)th entry from it.

5.3.4 Write Complexity

For each ORAMWrite operation, our write-only ORAM reads \( k \) random blocks, checks if they are empty, fills from zero to \( k - 1 \) of them with blocks from our stash, and updates the map accordingly.

The initial reading of the \( k \) blocks from the map (and hence checking if a block is empty) requires reading one block from each of \( \log n \) recursive levels. Therefore, we can accomplish
that first step with \(k \cdot O(B \cdot \log n) = O(B \cdot \log n)\) communication complexity. However, when we then write to the map, each recursive ORAM has to, itself, read from all the recursive ORAMs “below” it in order to read its own map. Thus, these writes costs \(O(B \cdot \log^2 n)\), bringing the total complexity for an ORAMWrite to \(O(B \cdot \log^2 n)\).

This overhead would be disappointing, as there are several full-functionality (read and write) ORAMs which provide the same overhead. However inspired by Stefanov et al. [7], we can use the following optimization to reduce the cost of the expensive map accesses. If we set the size of only the data blocks on our disk to \(B = \Omega(\log^2 n)\), the total size of \(\log n\) blocks in the recursive map is no greater than the size of one \(B\)-sized data block at the “top level”. What we end up with is non-uniform block sizes: top level blocks (actual data blocks) have size \(\Omega(\log^2 n)\), and blocks that are part of the map have only size \(\chi \log n\) for some constant \(\chi \geq 2\). We are still guaranteed that the map will have \(O(\log n)\) levels, but we reduce the communication complexity of a map operation by a factor of \(O(\log n)\). Consequently, in terms of communication complexity, reading from the map is constant in \(O(B)\), and updating/writing is in \(O(B \cdot \log n)\). We can apply the same optimization again, as long as \(B = \Omega(\log^3 n)\). In terms of communication for an ORAMWrite, we reduce total complexity to \(O(B)\).

In conclusion, our write-only ORAM features \(O(B \cdot s)\) memory and constant \(O(B)\) communication complexity.

### 5.3.5 Security Analysis

Security for our scheme follows directly from the fact that we only write to uniformly randomly chosen blocks at each level of the ORAM. Since the blocks we choose are independent of the addresses in the user’s access pattern, they cannot reveal any information about it to the adversary. Additionally, all the data we write is freshly encrypted with a semantically secure encryption, so the data itself cannot reveal any information.

**Simulator:** We note in the previous section that, to be useful in our scheme, an ORAM needs to be (efficiently) simulatable. Fortunately, such a simulator \(S\) is simple to construct for our scheme. \(S\) proceeds in every operation to change \(k\) uniformly random blocks at each level of the recursion to fresh random strings. Since, in normal operation of our ORAM, we will access \(k\) random blocks at each level, and those blocks will be filled with either random strings or IND$^\$-CPA encryptions indistinguishable from random, \(S\) will be indistinguishable from an actual execution of our ORAM.
5.4 Practical Hidden Volume Encryption with HIVE

Thus far we have presented a generic hidden volume encryption scheme which uses a write-only ORAM as a building block and has constant communication complexity per access. We now present HIVE that builds upon this idea and makes it practical. We start by addressing an important consideration that we must take into account when designing a practical, real-world system.

5.4.1 Uniform vs. Non-uniform Blocks

Current storage devices, e.g., today’s hard disks, have fixed size blocks (sectors). This means that we cannot use our non-uniform block optimization from the previous section, unless we wanted to use the base device blocks as “small” blocks and combine many of these blocks together to make “large” blocks. However, most systems use either 512 or 4096 byte blocks, so there is not much room for optimizing these parameters.

Fortunately, although we cannot easily obtain optimal $O(B)$ complexity with uniform blocks, our write-only scheme is still substantially more efficient than the currently most efficient full-functionality ORAMs, e.g., Path ORAM [7]. In Figure 5.2, we show the comparative costs for our scheme and Path ORAM for a concrete selection of parameters. Our write-only ORAM is more than an order of magnitude more efficient than Path ORAM.

To see why we are more efficient, it is useful to consider the complexity of our write-only ORAM and Path ORAM in terms of the level of recursions required to store the map. We denote this required level of recursion with $L$. In the uniform block setting, Path ORAM has $O(B \cdot L \cdot \log N)$ communication complexity. Since $L = O(\log n)$, this leads to the overall complexity of $O(B \cdot \log^2 N)$. Our write-only ORAM, by comparison, has $O(B \cdot L^2)$ complexity, with no independent $\log n$ factor. Again, since $L = O(\log n)$ our scheme has overall $O(B \cdot \log^2 N)$ complexity. However, for $L$ to actually approach the worst-case $\log n$, the block size needs to be close to $2 \cdot \log N$. In other words, with 512 byte = 4096 bit blocks, we would need a disk holding $2^{2056}$ bytes to approach that many levels of recursion ($4096 = 2 \cdot \log n$, so $N = 2^{2048}$ blocks, each of size $B = 512$ byte. Note that our write-only ORAM wastes 50% of all blocks, though). In contrast, up to 16 TB requires only $L \leq 3$ levels of recursion. However, for that same disk, we have $\log N = 35$. Therefore, for practical parameter choices, $L^2$ will be significantly smaller than $L \cdot \log N$, resulting in large savings of at least an order of magnitude.

The “jump” at 16 GByte ORAM size in Figure 5.2 is due to an increased level of recursion $L$ required for ORAMs of this size.
5.4.2 HIVE: Combining Volumes

A significant drawback of our generic hidden volume construction is that it requires a separate ORAM for each volume. This requires us to do a full operation on each of those ORAMs, resulting in a complexity dependent on $\max$. If we make a reasonable choice, say $\max = 10$, this could be a significant overhead. Fortunately, our ORAM construction is particularly suited to solving this problem. As we are only actually changing one data block each operation, independent of the number of volumes (the rest are reencryptions), we can thus combine all of our volumes into one. This is the main idea of our new scheme HIVE.

**Overview:** HIVE’s approach will be to store all the volumes randomly interleaved. Then, as shown in Figure 5.3, after writing to a block $b$, we will randomly change $b$’s mapping $\text{Map}[b]$ from $\beta$ to $\beta'$ using our write-only ORAM trick. This hides the write pattern for all volumes. We now present the full procedure for HIVE’s writing (HVEWrite) in Algorithm 12 and for HIVE’s reading (HVERead) in Algorithm 13.

**HVEWrite:** Compared with our original ORAM scheme, there is only one major change: instead of only modifying one block out of $k$ given by the indices in $S$, we now have to potentially modify all $k$ blocks. To see why, it is useful to consider the following scenario. Imagine user
\( U \) wants to write a block into \( V_1 \). In order to do that, they randomly choose \( k \) blocks to form \( S \). If, for instance, three of these blocks contain data from \( V_2 \), and one is empty, we cannot simply write our block from \( V_1 \) into this empty space because, after continuously writing “around” blocks in \( V_2 \), \( \mathcal{A} \) could infer its presence by the pattern we make trying to avoid it. Therefore, in order to be secure, we have to make sure that no previous writes to a \( V_j, j > i \) can influence a write to \( V_i \). Intuitively, if operations in higher volumes cannot influence operations in lower volumes, then the existence of a higher volume cannot be inferred from the observed behavior in a lower volume.

This can be accomplished by structuring the stash as a series of multiple queues (called \( \text{Stash}_i \) in Algorithm 12), one for each volume \( V_i \). For an HVEWrite, all the blocks in \( S \) are read and their content copied into the respective queues for their volume. A block on the disk may belong to some volume \( i \), or it may be a random string not containing any information. To put it into the correct queue, we have to attempt to decrypt the block with every key until we find the right volume or we run out of keys. This means that our ciphertexts must contain some redundant information, for decryption verification, but this can easily be accomplished by padding our plaintexts with a number of zero bits proportional to security parameter \( s \). Verification then consists of decrypting and checking if the plaintext begins with a sufficient number of zeroes. This meshes with our notion of security, because \( \mathcal{A} \), knowing some number of keys, cannot tell whether a block is part of a volume which he does not have the key for or if it is simply a random string.

Now, having freed \( k \) blocks on the disk, we will write \( k \) blocks back to the positions given by \( S \). For this, we read out of our stash queues, giving priority to the queues for lower volumes. That is, we empty the queue \( \text{Stash}_1 \) for \( V_1 \) first, followed by \( \text{Stash}_2 \) for \( V_2 \), etc. If there are less than \( k \) blocks in all stashes, then the leftover blocks from \( S \) are filled with random strings.

**HVERead**: For a HVERead, we read the block just like in our regular ORAM by querying the recursive map for the address and then reading that block from the disk. After we retrieve the target block, we also perform a “dummy” write to make reads and writes look identical to adversary \( \mathcal{A} \). This write does not change any values in the system, but it gives us a chance to write items from the stash, if necessary. Note that a block \( b \) we want to read in volume \( V_i \) might reside in the stash, so we first lookup \( b \) in \( \text{Stash}_i \).

**HVESetup**: Initialization is identical to our base ORAM, we simply initialize the map to an “empty” state where every block starts unmapped.

**Complexity**: Figure 5.4 shows how HIVE is more efficient than the generic construction using \( \text{max} \) separate ORAMs. Although HIVE performs better for a large range of parameters, note that HIVE scales exponentially in \( L \). Since we have to change up to \( k \) blocks at each level, which in turn requires changing \( k \) blocks in all the levels below them, the overall complexity is \( O(k^L) \).
With \( \text{max} = 10 \), for up to 1 Exabyte, HIVE is still cheaper than our generic construction since it is independent of \( \text{max} \). However, with larger volumes and more levels of recursion, HIVE becomes less practical than our first construction. Therefore, depending on the choice of parameters, it can be more efficient to use one or the other.

For completeness sake, we note that Path ORAM can also be modified to produce a “combined volume” version in a similar manner (due to its use of a stash), but it would be significantly less efficient than our construction, just as Path ORAM is less efficient than our write-only ORAM.

**Security:** Since a block from \( V_j, j > i \) is written only if the queue for \( V_i \) is already empty, blocks from \( V_j \) cannot influence \( A \)'s view of \( V_i \). Additionally, since an encryption under \( P_j \) is indistinguishable from a random string to \( A \) which does not know \( P_j \). What this means is that \( A \)'s view of the disk cannot be impacted by volumes which he does not have the key to, therefore we achieve \( \Gamma^{\text{Arbitrary, Plausible}}_{A, \Sigma} \) security as before.

**HIVE-B:** To make HIVE secure against Opportunistic adversaries, we can follow the same idea as in Section 5.3.2. Instead of immediately writing in volumes \( V_i, i > 1 \), we just add the block to Stash\(_i\). When we do operations in \( V_1 \), we proceed as normal, writing as much as we can from Stash\(_2\), Stash\(_3\), etc.

### 5.4.3 Discussion

We have shown that, under various adversarial models, HIVE does not give any information about the number of volumes in use, beyond what is known a priori. Yet, in practice, just the presence of HIVE on a user’s system is an indication that there might be more than one volume.

However, we stress that there is a legitimate reason a user would want to use HIVE with only a single volume: even with a single volume, HIVE offer stronger security than related work.
such as TrueCrypt. It provides security against multiple snapshot adversaries that could deduce information from the user’s access pattern. For instance, between snapshots adversary $A$ may see a particular file has been written that matches the size of a known file. $A$ does not need the decryption key to determine what the file is. Therefore, even without the encryption key, significant data leakage can occur to a multiple snapshot adversary just by observing patterns of changes.

This gives a plausible reason why a user would be using only a single volume and hence allows for deniability in the case that they actually do have hidden volumes.

Since HIVE acts as an Oblivious RAM for each of the volumes, the user’s access pattern is hidden. Given our new write-only ORAM, our approach is more efficient than simply using existing ORAMs not tailored for disk encryption.

### 5.5 Implementation

To show its real-world practicality, we have implemented HIVE for Linux. Our implementation comprises a kernel module offering a virtual block device for each volume and a userland tool to manage these volumes. The source code is available for download [46].

The kernel module is built using *device-mapper*, a standard Linux kernel framework for mapping block devices onto virtual devices, also used to implement technologies such as LVM, dm-crypt and software RAID.

Device-mapper allows placing HIVE between the Linux block IO layer and the underlying device drivers. There, HIVE intercepts all block IO requests in flight, splits them into single-block-sized chunks, remaps them to their new physical blocks on the disk, and performs cryptography operations, as previously described. Note that our implementation works on any block device (e.g., hard disks, USB sticks, network block devices, etc.) since it stacks on top of the actual device driver which communicates with hardware.

We use AES-CBC with 256 Bit keys for encryption and PBKDF2 for key derivation. For performance reasons, we generate randomness using RC4, using the kernel’s entropy pool only to generate an initial key for RC4. Our implementation supports up to a 4 KB logical block size (this limit is imposed by the x86 architecture and kernel internals), regardless of the underlying hardware’s physical structure. In our evaluation presented below, we set the block size $B$ to 4 KB, even though our test device has 512-byte sectors. This minimizes the number of random disk accesses performed during IO and results in a significant performance improvement. As
Table 5.2: HIVE Benchmarks, $L = 2, k = 3$

<table>
<thead>
<tr>
<th></th>
<th>Seq. Write (MB/s)</th>
<th>Seq. Read (MB/s)</th>
<th>Create (Kfiles/s)</th>
<th>Stat (Kfiles/s)</th>
<th>Delete (Kfiles/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw disk</td>
<td>216.04</td>
<td>221.74</td>
<td>82.29</td>
<td>201.18</td>
<td>105.10</td>
</tr>
<tr>
<td>HIVE</td>
<td>0.97</td>
<td>0.99</td>
<td>1.57</td>
<td>3.23</td>
<td>1.79</td>
</tr>
</tbody>
</table>

another performance optimization, our system disables IO reordering and scheduling in the kernel for the virtual devices, because HIVE always performs random device access and cannot benefit from kernel’s access pattern anticipation features.

The userland tool allows users to create, mount and unmount HIVE devices `/dev/mapper/HIVEi` for volume $V_i$ on top of any other block device (e.g., `/dev/sda`). Upon receiving the create command, our tool formats the specified device by creating the necessary metadata structures, such as the $\text{max}$ different Maps, $\text{max}$ stashes of fixed size, IVs and the reverse mappings for data blocks. Note that our implementation allows for recursion, so it recursively stores and accesses the Maps of fixed size as described in Section 5.3.3. Finally, to mount or unmount these volumes, our tool issues the appropriate ioctl commands to the kernel’s device-mapper module.

**Benchmarks:** We have tested our implementation on a standard desktop computer with an Intel i7-930 CPU, 9 GB RAM (although RAM was not an issue during our evaluation), running Arch Linux x86-64 with kernel 3.13.6. As the underlying block device, we have used an off-the-shelf Samsung 840 EVO SSD.

For the evaluation, we used bonnie++, a standard disk and filesystem benchmarking tool. Note that in the face of IO caching by the OS, files created in the bonnie++ benchmarks must be set to twice the size of system memory installed (9 GB in our case) to reliably measure device performance. To speed up the total benchmark time, we modified bonnie++, flushing IO buffers to the device after running a benchmark, and signaling the kernel to drop page, dentry, and inode caches before the next run. This ensured that our performance measurements remained unaffected from caching.

We have first tested an ext4 filesystem with 4 KB blocks on the “raw” disk to get a baseline. We then created 2 hidden volumes on our disk and set $L = 2$ and $k = 3$. With this configuration, HIVE supports volumes of a total size of up to 16 TByte. We repeated the experiments by running bonnie++ on an ext4 filesystem created on top of the HIVE block device. Table 5.2 presents the results, averaged over 5 runs with a relative standard deviation of $< 6\%$. These results show that IO operations (sequential writes and reads in MB per second) were slower by a factor of $\approx 200$, while filesystem operations (create, stat, and delete in thousands of files per second) were slower by a factor of 50 to 60. Random seek performance was not measurable on the raw SSD (i.e., bonnie++ reported that the tests completed too quickly to measure reliable timings),
whereas HIVE achieved 1.2 Kseeks/s. The HIVE induced CPU utilization was low with < 1\% during measurements, indicating that random access IO constitutes the main bottleneck.

We conclude that, while the slowdown is certainly significant, a throughput of 1 MB/s on an off-the-shelf disk is acceptable in many scenarios, rendering HIVE practical for the real-world. Future work is dedicated for additional performance tuning.

5.6 Conclusion

In this chapter, we have proposed new, parameterized definitions of formal security for hidden volume encryption. Existing work lacks security against strong adversaries with multiple snapshot and chosen plaintext capabilities. We have proposed new constructions which meet strong security definitions using ORAM as a building block. Observing that strong security can be provided with less powerful write-only ORAM, we have then introduced a novel construction of a write-only ORAM which achieves optimal $O(1)$ communication complexity. This is a surprising result, indicating that writes are easier to hide than reads. Additionally, we have shown how our ORAM can be specially adapted to the problem of hidden volume encryption to produce an even more efficient solution, which we dub HIVE. Finally, we have implemented our scheme as a kernel-level block device and benchmarked its performance on commodity hardware, achieving a throughput of $\approx 1$ MByte/s.
Input: Block index $b$, data $d$, volume $i$, passwords $\mathcal{P}$

if $b \neq i$ then
    Enqueue($\text{Stash}_i$, $(b, d)$);
end

$S := \{ \beta_j \},$ such that $\beta_j \in \{ 1, \ldots, N \} \land 1 \leq j \leq k \land \forall u, v : \beta_u \neq \beta_v \}$ holds;

// Fetch blocks from $S$ and put into stashes
for $u := 1$ to $k$
    $d := \text{DiskRead}(S[u])$;
    if $d$ is block $b$ from volume $V_v$ then
        // Derive $\kappa_v$ from $P_v$
        $d := \text{Dec}_{\kappa_v}(d)$;
        Enqueue($\text{Stash}_v$, $(b, d)$);
    end
end

$v := 1$;
for $u := 1$ to $\max$
    if $v \leq k \land \text{Stash}_u \neq \emptyset$ then
        $(b, d) := \text{DeQueue}(\text{Stash}_v)$;
        DiskWrite($S[v]$, $\text{Enc}_{\kappa_v}(d)$);
        $v := v + 1$;
    end
end

while $v \leq k$
    // Fill remaining blocks in $S$ with random strings
    $r \in \{ 0, 1 \}_B$;
    DiskWrite($S[v]$, $r$);
end

$M := \left\lfloor \frac{B}{\log N} \right\rfloor$;
for $j := 1$ to $k$
    if $S[j]$ was filled with a real block then
        // Let $b'$, $v'$ be the block index and volume number of the block that was written to disk block $S[j]$
        $\text{map}_{\text{block}} := \text{MapRead}(i', \left\lfloor \frac{B}{M} \right\rfloor)$;
        $\text{map}_{\text{block}}[b' \mod M] := S[j]$;
        Map.HVEWrite($i', \left\lfloor \frac{B}{M} \right\rfloor$, $\text{map}_{\text{block}}$);
    else
        // Do a dummy operation to the map
        Map.HVEWrite($\perp, \perp, \perp, \perp$);
    end
end

Algorithm 12: HIVE HVEWrite($b, d, i, \mathcal{P}$)

Input: Volume $i$, block index $b$
if block $b$ is in Stash, then
    Read and return most recent version of block $i$ from Stash$_i$;
end

$M := \left\lfloor \frac{B}{\log N} \right\rfloor$;
$\text{map}_{\text{block}} := \text{Map.HVERead}(i, \left\lfloor \frac{B}{M} \right\rfloor)$;
$\text{location} := \text{map}_{\text{block}}[b \mod M]$;
$d := \text{DiskRead}(\text{location})$;
// Do a “dummy” write
HVEWrite($\perp, \perp, \perp, \perp$);
return $\text{Dec}_{\kappa_i}(d)$

Algorithm 13: HIVE HVERead($b, i$)
Chapter 6

Multi-client Oblivious RAM

Practicality of ORAMs has improved significantly in the last few years, as shown in previous chapters. However, at least one significant hurdle to overcome for actual adoption remains: security of modern ORAMs relies on there being only a single client at all times. The problem stems from the fact that, in order to hide the access pattern, ORAM algorithms must modify some of the data on the server after every access. If the server is permitted to, e.g., “rewind” the data and present an old version to a client, further interactions may reveal something about their access pattern. Fortunately, with a single client this is easily solved by storing a small token, such as the root of a hash tree [11]. This token authenticates and verifies freshness of all data retrieved from the server, ensuring that no such rewind attack is possible.

In this chapter, we address some more complex scenarios where multiple clients share data stored in a single ORAM. With multiple clients, an authentication token is not sufficient. Data may not pass one client’s authentication for a valid reason: it has been modified by one of the other clients. If clients could communicate with each other using a secure out-of-band channel, then it becomes possible to continually exchange and update each other with the most recent token. However, existence of secure out-of-band-communication is not always a reasonable assumption. If clients already have a secure method of continuously communicating with each other, one may argue that ORAM may not even be needed in the first place. Current solutions for multi-client ORAM work only in the presence of an honest-but-curious adversary, which cannot perform rewind attacks on the clients. Often, this is not a very satisfying model, since rewind attacks are very easy to execute for real-world adversaries and would be difficult to detect. Goodrich et al. [35], in their paper examining multi-client ORAM, recently proposed as an open question whether one could be secure for multiple clients against a malicious server.

**Technical Highlights:** In this chapter, we introduce the first construction for a multi-client ORAM. We prove security even if the server is fully malicious. Our contribution is twofold, specifically:
First, we focus on two ORAM constructions that follow a “classical” approach, the square-root ORAM by Goldreich [10] and the hierarchical ORAM by Goldreich and Ostrovsky [3]. We adapt these ORAMs for multi-client security. Our approach is to separate client accesses into two parts. One part can be performed securely in the presence of a malicious server. The other part cannot be performed securely, but contains an efficient integrity check which will reveal any malicious behavior, thereby allowing the client to terminate the protocol. Roughly speaking, we replicate the ORAM for each client, such that clients read only from their copy of the ORAM (integrity protected), but write into all ORAM copies (trivially secure).

The “classical” ORAM constructions have been largely overshadowed by more recent tree-based ORAMs. Tree-based ORAMs such as the one by Shi et al. [5] and Stefanov et al. [7] (Path ORAM) and many derivatives, provide better efficiency and worst-case guarantees. Consequently, we go on to demonstrate how a multi-client secure Path ORAM can be constructed. Among changing stash behavior and address mapping, we solve the key challenge of realizing a multi-client secure ReadAndRemove by storing Path ORAM’s metadata using small “classical” ORAMs as building blocks. For reasonably small block sizes, this results in a multi-client ORAM which has the same per-client communication complexity as current single-client ORAMs: $O(\log n)$.

6.1 Multi-client ORAM

Instead of a single ORAM accessed by a single client, we can envision multiple clients securely exchanging or sharing data stored in a single ORAM. For example, imagine multiple employees of a company that want to read from and write into the same database stored at an untrusted server. Similar to standard ORAM security, sharing data and jointly working on the database should not leak the clients’ access patterns to the server. Alternatively, we can also envision a single client with multiple different devices (laptop, tablet, smartphone) accessing the same data hosted at an untrusted server. Again, working on the same data should not reveal access patterns. Throughout this chapter we consider the terms “multi-client” and “multi-user” to be equivalent. Our model assumes that the clients all trust each other, and so “multi-client” may be a more intuitive description, but we hope that this research can be expanded in the future to accommodate more fine-grained security.

ORAM protocols provide security because they are highly stateful. In order to hide the fact that a client accesses a certain data block, ORAMs typically perform shuffling or reordering of blocks so that two accesses are not recognizable as being the same. An obvious attack that a malicious server can do is to undo or “rewind” that shuffling after the first access and present the same, original view of the data to the client when they make the second access. If the client
was to blindly execute their access, and it was the same block of data as the first access, it would result in the same pattern of interactions with the server that the first access did. The server would immediately have broken the security of the scheme. This is a straightforward attack, and it is easily defeated by having the client store a token for authentication and freshness [11].

However, with two (or more) clients sharing data in an ORAM, the server can execute the same attack, but against the two clients separately. After watching one client retrieve some data, he can rewind the ORAM’s state and present the original view to the second client. If the second client accesses the same data that the first client did, the server will recognize it and break security. Without having some secure side-channel to exchange authentication tokens after every access, it is difficult for clients to detect such an attack.

Note that against an honest-but-curious adversary multi-client security is trivial. The adversary is guaranteed not to change any data on the server and so any ORAM protocol with small client memory can be used with multiple clients. The clients simply upload all of their local memory to the server after every access (encrypted of course) and the next client downloads it to continue the protocol where the first client left off. This approach is not feasible against a malicious server because the adversary can tamper with the client memory or the structure of the data as described above. Therefore the malicious adversary is crucial to the motivation for our work.

### 6.1.1 Security Definition

We start by briefly recalling the standard Oblivious RAM concept. An ORAM provides an interface to read from and write to blocks of a RAM (an array of storage blocks). It supports Read\((x)\), to read from the block at address \(x\) and Write\((x, v)\) to write value \(v\) to block \(x\). The ORAM allows storage of \(N\) blocks, each of size \(B\). To securely realize this functionality, an ORAM outsources a state \(\Sigma\) to an untrusted storage. For convenience, state \(\Sigma\) can be represented as a sequence of fixed-length strings. We will call the untrusted storage provider a server here because the most likely application for a multi-client ORAM would be outsourced cloud storage.

**Definition 6.1** (ORAM Operation \(\text{OP}\)). An operation \(\text{OP}\) is defined as \(\text{OP} = (o, x, v)\), where \(o = \{\text{Read}, \text{Write}\}\), \(x\) is the virtual address of the block to be accessed and \(v\) is the value to write to that block. \(v = \bot\) when \(o = \text{Read}\).

We now present our multi-client ORAM security definition which slightly augments the standard, single-client ORAM definition.

**Definition 6.2** (Multi-client ORAM \(\text{II}\)). A multi-client ORAM \(\text{II}\) is defined by tuple \(\text{II} = (\text{Init, Access})\).
1. Init($\lambda, N, B, \psi$) initializes $\Pi$. It takes as input security parameter $\lambda$, total number of blocks $N$, block size $B$, and number of clients $\psi$. Init outputs an initial ORAM state $\Sigma_{\text{init}}$, which encompasses the entirety of the ORAM that is stored on the server, and a list of per client states $\{st_{u_1}, \ldots, st_{u_\psi}\}$ which are kept local to the individual clients.

2. Access($OP, \Sigma, st_{u_i}$) performs operation $OP$ on ORAM state $\Sigma$ using client $u_i$’s state $st_{u_i}$. Access outputs (1) an access pattern $< (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) >$, where $(\alpha_j, \nu_j)$ denotes that the string at position $\alpha_j$ in state $\Sigma$ is read from or replaced by string $\nu_j$, and (2) a new state $st_{u_i}$ for client $u_i$.

In contrast to single-client ORAM, a multi-client ORAM introduces the notion of clients. This is modeled by different per-client states, $st_{u_i}$ for client $u_i$. Algorithm Init outputs different initial states $st_{u_i}$ and, in practice, would distribute these initial states to clients. One can assume this distribution during initialization taking place over a secure out-of-band communication channel. However after initialization, clients cannot use an out-of-band channel anymore. Whenever client $u_i$ executes Access on the multi-client ORAM, they can only update their own state $st_{u_i}$.

Finally, we define the security of a multi-client ORAM against malicious servers. Consider experiment $\text{Sec}_{\lambda, \Pi}^{\text{ORAM}}(\lambda)$ below.

**Definition 6.3 (Multi-client ORAM $\text{Sec}_{\lambda, \Pi}^{\text{ORAM}}(\lambda)$).**

\[
b \xleftarrow{\$} \{0, 1\} \\
(\Sigma_{\text{init}}, st_{u_1}, \ldots, st_{u_\psi}) \leftarrow \text{Init}(\lambda, n, B, \psi) \\
(\Sigma, OP_0, OP_1, i, st_A) \leftarrow A(\lambda, n, B, \Sigma_{\text{init}}, \psi) \\
\text{for } j = 1 \text{ to poly}(\lambda) \text{ do} \begin{align*}
& (st_{u_i}, < (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) >) \leftarrow \text{Access}(OP_b, \Sigma, st_{u_i}) \\
& (\Sigma, OP_0, OP_1, i, st_A) \leftarrow A(< (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) >, st_A)
\end{align*} \\
b' \leftarrow A(st_A) \\
\text{output } 1 \text{ iff } b = b'
\]

An ORAM $\Pi = (\text{Init}, \text{Access})$ is multi-client secure if for all PPT adversaries $A$

\[
Pr[\text{Sec}_{\lambda, \Pi}^{\text{ORAM}}(\lambda) = 1] < \frac{1}{2} + \epsilon(\lambda),
\]

where $\epsilon$ is a negligible function in security parameter $\lambda$.

First, a random bit $b$ is chosen, and both the ORAM and adversary $A$ are initialized. Then, $A$ gets oracle access to the ORAM and can adaptively query it during $\text{poly}(\lambda)$ rounds. In each round, $A$ selects a client $u_i$, determines two operations $OP_0$ and $OP_1$, and outputs an ORAM state $\Sigma$. The oracle performs operation $OP_b$ as client $u_i$ with state $st_{u_i}$ and ORAM state $\Sigma$.
using protocol II. The oracle returns access pattern $(\alpha_i, \nu_i)$ induced by II back to $\mathcal{A}$. Each tuple $(\alpha_i, \nu_i)$ tells the adversary which part of $\Sigma$ was read or overwritten (with value $\nu$). Eventually, $\mathcal{A}$ guesses $b$.

Our game-based definition is equivalent to ORAM’s standard security definition with two exceptions: we allow the adversary to arbitrarily change the state of the on-server storage $\Sigma$, and we split the ORAM algorithm into $\psi$ different pieces which cannot share state among themselves.

Note that, in this work, we assume that all clients trust each other and do not conspire. For ease of exposition, we assume that all client share a key $\kappa$ used for encryptions, decryptions, and MAC computations that we will introduce later.

**Consistency:** The adversary we assume in this chapter is fully malicious (in contrast to, e.g., semi-honest) and learning access patterns. However, there is one caveat: while certainly important, we stress that consistency issues are a non-goal in this chapter. Informally, $\mathcal{A}$ could present different versions of ORAM state $\Sigma$ to different clients, leading to desynchronization and inconsistent views of the ORAM. Not surprisingly, it is impossible to protect against desynchronization in the absence of out-of-band communication. The strongest consistency achievable would be *fork* consistency. For more information, see Li et al. [59]. In this chapter, we allow $\mathcal{A}$ to desynchronize clients, as long as security following Definition 6.3 is not violated. That is, clients’ access patterns must never be revealed. In accordance with related work, we also do not consider “reaction-attacks” [60] (and variants). In a reaction attack, $\mathcal{A}$ would would observe the client’s reaction to, e.g., a failed integrity check. For example, sending a forged data to a client could force the client’s higher applications layers into accessing the ORAM more (or less) often. Reaction attacks are a general problem with cryptographic protocols and are not considered here.

### 6.2 Multi-client Security for Classical ORAMs

We start by demonstrating how two existing ORAM constructions, the original square-root solution (presented in section 1.1.1) by Goldreich [10] and the hierarchical one by Goldreich and Ostrovsky [3], can be transformed into multi-client secure versions with the same complexity per client.

**Challenge** When considering a multi-client scenario, it becomes very easy for a malicious server to break security of the square-root ORAM. For example, client $u_1$ can access a block $x$ that is not in the cache, requiring $u_1$ to read $\pi(x)$ from main memory and insert it into the cache.
The malicious server now restores the cache to the state it was in before \( u_1 \)'s access added block \( x \). If a second client \( u_2 \) also attempts to access block \( x \), the server will now observe that both clients read from the same location in main memory and know that \( u_1 \) and \( u_2 \) have accessed the same block. Without the clients having a way to communicate directly with each other and pass information that allows them to verify the changes to the cache, the server can always “rewind” the cache back to a previous state. This will eventually force one client to leak information about their accesses.

**Rationale** Our approach for multi-client security is based on the observation that the *cache update* part of the square-root solution is secure by itself. Updating the cache only involves downloading the cache, changing one element in it, re-encrypting, and finally storing it back with the server. Downloading and later uploading the cache implies always “touching” the same \( \sqrt{n} \) blocks, independently of what the malicious server presents to a client as \( \Sigma \), and also independent of the block being updated by the client. Changing values inside the cache cannot leak any information to the server, as its content is always newly IND-CPA encrypted. Succinctly, being similar to a trivial ORAM, updating a cache is automatically multi-client secure.

However, any reading can leak information. Reading from main memory is conditional on what the client finds in the cache. We call this part the *critical* part of the access, and the cache update correspondingly *non-critical*. To counteract this leakage, we implement the following changes to enable multi-client support for the square-root ORAM:

1. **Separate ORAMs**: Instead of a single ORAM, we use \( \phi \) separate ORAMs, \( \text{ORAM}_1, \ldots, \text{ORAM}_\phi \), one for each client. Each client \( u_i \) will perform the critical part of the access only on their own ORAM, that is, \( \text{ORAM}_i \)'s main memory and cache. Thus, each client can guarantee they will not read the same address from their ORAM’s main memory twice. However, any change to the cache as part of ORAM Read\((x)\) or Write\((x, v)\) operations will be written to every ORAM’s cache. Updating the cache on any ORAM is already guaranteed to be multi-client secure and does not leak information.

2. **Authenticated Caches**: For each client \( u_i \) to guarantee that they will not repeat access to the main memory of \( \text{ORAM}_i \), the cache is stored together with an encrypted access counter \( \chi \) on the server. Each client stores locally a MAC over both the cache and the encrypted access counter \( \chi \) of their own ORAM. Every access to their own cache increments the counter and updates the MAC. Since clients read only from their own ORAMs, and they can always verify the counter value for the last time that they performed a read, the server cannot roll back beyond that point. Two reads will never be performed with the cache in the same state.
Input: Security parameter $\lambda$, number of blocks in each ORAM $n$, block size $B$, number of clients $\phi$
Output: Initial ORAM state $\Sigma_{\text{init}}$, initial per client states $\{s_{t_1}, \ldots, s_{t_\phi}\}$

$\kappa \leftarrow \{0, 1\}^\lambda$;

for $i := 1$ to $\phi$ do
  Generate permutation $\pi_{i,0}$ from key $\kappa$;
  Initialize $\sqrt{N} + n$ main memory blocks, shuffled with $\pi_{i,1}$, and $\sqrt{N}$ cache blocks;
  Set cache counter $\chi_i = 0$;
  Set epoch counter $\gamma_i = 0$;
  $\text{ORAM}_i = \text{Enc}_\kappa(\text{main memory}) \longrightarrow \text{Enc}_\kappa(\text{cache} \longrightarrow \chi_i \longrightarrow \gamma_i)$;
  $\text{mac}_i = \text{MAC}_\kappa(\text{Enc}_\kappa(\text{cache} \longrightarrow \chi_i \longrightarrow \gamma_i))$;
  Send $s_{t_i} = \{\kappa, \chi_i\}$ to client $u_i$;
end
Send $\Sigma_{\text{init}} = \{(\text{ORAM}_1, \text{mac}_1), \ldots, (\text{ORAM}_\phi, \text{mac}_\phi)\}$ to server;

**Algorithm 14:** $\text{Init}(\lambda, n, B, \phi)$, initialize ORAM

### 6.2.1 Details

We detail the above ideas in two algorithms: Algorithm 14 shows the initialization procedure, and Algorithm 15 describes the way a client performs an access with our multi-client secure square-root ORAM.

First, we introduce the notion of an `epoch`. After $\sqrt{N}$ accesses to an ORAM, its cache is “full”, and the whole ORAM needs to be re-shuffled. Re-shuffling requires computing a new permutation $\pi$. Per ORAM, a permutation can be used for $\sqrt{N}$ operations, i.e., one `epoch`. The next $\sqrt{n}$ operations, i.e., the next epoch, will use another permutation and so on. In the two algorithms, we use an epoch counter $\gamma_i$. Therewith, $\pi_{i, \gamma_i}$ denotes the permutation of client $u_i$ in ORAM$_i$’s epoch $\gamma_i$. For any client, to be able to know the current epoch of ORAM$_i$, we store $\gamma_i$ together with the ORAM’s cache on the server.

On a side note, we point out that there are various ways to generate pseudo-random permutations $\pi_{i, \gamma_i}$ on $n$ elements in a deterministic fashion. For example, one can use $\text{PRF}_\kappa(i \mid| \gamma_i)$ as the seed in a PRG and therewith perform a Fisher-Yates shuffle.

In addition to the epoch counter, we also introduce a per client `cache counter` $\chi_i$. Using $\chi_i$, client $u_i$ counts the number of accesses of $u_i$ to the main memory and cache of their own ORAM$_i$.

After each access to ORAM$_i$ by client $u_i$, $\chi_i$ is incremented. Each client $u_i$ keeps a local copy of $\chi_i$ and therewith verifies freshness of data presented by the server. As we will see below, this method ensures multi-client ORAM security. Note in Algorithm 14 that a client $u_j$ never increases $\chi_i$ of another client $u_i$. Only $u_i$ ever updates $\chi_i$.

In our algorithms, $\text{Enc}_\kappa$ is an IND-CPA encryption such as AES-CBC. For convenience, we only write $\text{Enc}_\kappa(\text{main memory})$, although the main memory needs to be encrypted block by block to allow for the retrieval of specific blocks. Also, for the encryption of main memory blocks, $\text{Enc}_\kappa$ offers authenticated encryption such as encrypt-then-MAC.
Input: Address x, new value v, client ui, stui = {k, χi}
Output: The value of block x, new state stub = {k, χi}
From ORAMi: read ci = Enck(ci) and maci;
maci' = MACk(ci);
if maci' ≠ maci then
    output Abort;
end
Decrypt ci to get cache and counter χi';
if χi' < χi then
    output Abort;
end
if block x ∉ cache then
    Read and decrypt block πi,γi(x) from ORAMi’s main memory;
else
    Read next dummy block from ORAMi’s main memory;
end
if v = ⊥ then // operation is a Read
    ν ← v;
else // operation is a Write
    ν ← existing value of block x;
end
Append block (x, ν) to cache;
if cache is full then
    γi = γi + 1;
    Compute new permutation πi,γi;
    Read and decrypt ORAMi’s main memory;
    Shuffle cache and main memory using πi,γi;
    Send Enck(main memory) to server to update ORAMi;
end
χi = χi' + 1;
maci = MACk(Enck(cache —— χi —— γi));
Send new Enck(cache —— χi —— γi) and maci to server to update ORAMi;
for j ≠ i do // for all ORAMj ≠ ORAMi
    Read and decrypt cache and χj from ORAMj;
    Read and verify maci from ORAMj;  
    Append block (x, ν) to cache;
    if cache is full then
        γj = γj + 1;
        Compute new permutation πj,γj;
        Read and decrypt ORAMj’s main memory;
        Shuffle cache and main memory using πj,γj;
        Send Enck(main memory) to server to update ORAMj;
    end
    macj = MACk(Enck(cache —— χj —— γj));
    Send new Enck(cache —— χj —— γj) and macj to server to update ORAMj;
end
output (ν, stub = {k, χi});
Algorithm 15: Access(ORAMi, stui), perform Read or Write

A client can determine whether a cache is full in Algorithm 15 by the convention that empty blocks in the cache decrypt to ⊥. As long as there are blocks in the cache remaining with value ⊥, the cache is not full.

Init: All φ ORAMs together with their cache and epoch counters are initialized. The server
stores the ORAMs and MACs computed with a single key κ. Each client receives their state, comprising κ and cache counter.

Access: After verifying the MAC for ORAMᵢ and whether its cache is not from before 𝑢ᵢ’s last access, 𝑢ᵢ performs a standard Read or Write operation for block 𝑥 on ORAMᵢ. If the cache is full, 𝑢ᵢ re-shuffles ORAMᵢ updating π. In addition, 𝑢ᵢ also adds block 𝑥 to all other clients’ ORAMs. Note that for this, 𝑢ᵢ does not read from the other ORAMs, but only completely downloads and re-encrypts their cache.

Note on complexity In addition to the communication complexity involved, there is also computation the client must perform in our scheme. Fortunately, the computation is exactly proportional to the communication and easily quantifiable. Every block of data retrieved from the server has a MAC that must be verified and a layer of encryption that must be removed. Since modern ciphers and hash functions are very efficient, and can even be done in hardware on many computers, communication is the clear bottleneck. For comparison, encryption and MACs are common on almost every secure network protocol, so we consider only the communication overhead in our analysis.

6.2.2 Security Analysis

First, we ensure with Lemma 6.4 that once a block 𝑥ᵢ,ⱼ enters the cache of ORAMᵢ, it can never be removed without client 𝑢ᵢ noticing or the end of an epoch (and a new shuffle) occurring.

Lemma 6.4. Let Γᵢ,j be the state of the cache of ORAMᵢ when client 𝑢ᵢ executes their jth access. Let R(Γ, 𝑥) be the predicate 𝑥 ∈ Γ which indicates if block 𝑥 is already resident in the cache Γ. Let 𝑥ᵢ,j be the virtual block that client 𝑢ᵢ accesses during operation j. E(𝑖, 𝑗) is the epoch that ORAMᵢ is in (as represented by the data returned from the adversary) when client 𝑢ᵢ executes operation number 𝑗.

For all PPT adversaries 𝑨 and security parameter λ, there exists a negligible function 𝜖 such that

\[ \Pr[\text{If } R(Γᵢ,j, 𝑥ᵢ,j), \text{ then}] \]

\[ (\forall k > j \text{ with } E(𝑖, j) = E(𝑖, k) \land 𝑥ᵢ,j = 𝑥ᵢ,k) : \]

\[ R(Γᵢ,j, 𝑥ᵢ,k) \text{ or client } 𝑢ᵢ \text{ outputs } \text{Abort} = 1 - \epsilon(λ) \]

Proof. This follows from the security of MAC and the fact that no client will remove a block from the cache unless they are performing a shuffle. If during an access 𝑗 client 𝑢ᵢ sees a counter
value greater than or equal to the counter value from access $j - 1$, and the MAC verifies, then they can be sure that every element in the cache during access $j - 1$ is also in the cache during access $j$ (unless there was a shuffle between).

Now, Lemma 6.5 shows that if Lemma 6.4 holds, (Init, Access) is multi-client secure during a single epoch.

Lemma 6.5. For all PPT adversaries $A$ and security parameter $\lambda$, there exists a negligible function $\epsilon$ such that

$$\Pr[If \forall i, j, k:\]
\begin{align*}
x_{i,j} \neq x_{i,k} \lor R(\Gamma_{i,j}, x_{i,j}) \lor R(\Gamma_{i,k}, x_{i,k}) \\
\lor E(i, j) \neq E(i, k)] = 1 - \epsilon(\lambda)
\end{align*}$$

then (Init, Access) is a multi-client ORAM secure against malicious adversaries.

Proof. The writing part of a Read or Write operation always reads and writes the same strings $\alpha \in \Sigma$, namely the cache and its authentication data (counters and mac). Therefore, with IND-CPA encryption, any pair of operations $OP_0$ and $OP_1$ will be indistinguishable for the writing part.

The read part on the other hand contains a conditional access to main memory. The goal is to show that this access does not leak any information that would allow an adversary to distinguish between two accesses. Our condition above ensures that there will not be two operations in the same epoch where the client requests a block and it is not in the cache. Since a block in main memory is only accessed if it does not already exist in the cache, this guarantees that each client $i$ will never access the same block in main memory twice in the same epoch. Recall that every block is mapped to a random location, following a permutation $\pi$. If $\pi$ is a random permutation, then the access pattern to main memory will be indistinguishable from random accesses, and the adversary’s view will be indistinguishable for all pair of operations $OP_0$ and $OP_1$.

Theorem 6.6. For all PPT adversaries $A$ and security parameter $\lambda$, there exists a negligible function $\epsilon$ such that (Init, Access) is a multi-client Oblivious RAM secure against $A$ with probability $1 - \epsilon(\lambda)$.

Proof. In the same epoch, security follows from Lemma 6.5 and Lemma 6.4. Between epochs, main memory is re-shuffled and the ORAM is effectively reinitialized, with security of this new epoch being ensured as was the previous.
**Deamortizing** Goodrich et al. [61] propose a way to deamortize the classical square-root ORAM such that it obtains a worst-case overhead factor of $\sqrt{N} \cdot \log^2(N)$. Their method involves dividing the work of shuffling over the $\sqrt{n}$ operations during an epoch such that when the cache is full there is a newly shuffled main memory to swap in right away. This shuffling induces an access pattern in the RAM which is independent of the block a client is trying to access (it is performed along with the client request simply to spread the work out), and as such can also be incorporated into our scheme to achieve sublinear worst-case overhead.

**Multi-client security** As this scheme is a generalization of the square-root one, our modifications extend naturally to provide multi-client security. Again, each client should have their own ORAM which they read from. Writing to other clients’ ORAMs is done by inserting the block into the top level of their cache and then shuffling as necessary. The only difference this time is that each level of the cache must be independently authenticated. Since the cache levels are now hash tables, and computing a MAC over every level for each access would require downloading the whole data structure, we can instead use a Merkle tree [62]. This allows for efficient verification and updating of pieces of the cache without having access to the entire thing, and it maintains poly-log communication overhead.

### 6.3 Tree-based Construction

While pioneering the research, classical ORAMs have been outperformed by newer tree-based ORAMs which achieve better average and worst-case complexity. We now proceed to show how these constructions can be modified to also support multiple clients. Our strategy will be similar to before, but with one major twist: in order to avoid linear worst case complexity, tree-based ORAMs do only small local “shuffling,” which turns out to make separating a client access into critical and non-critical parts much more difficult. When writing, one must not only add a new version of the block to the ORAM, but also explicitly mark the old version as obsolete, requiring a conditional access. This is in contrast with our previous construction where old versions of a block would simply be discarded during the shuffle.

#### 6.3.1 Overview

For this section, we will use Path ORAM [7] as the basis for our multi-client scheme, but the concepts apply similarly to other tree-based schemes.

**Integrity** Because of its tree structure, it is straightforward to ensure integrity in Path ORAM. The client can store a MAC in every node of the tree that is computed over the contents of that
node and the respective MACs of its two children. Since the client accesses entire paths in the tree at once, verifying and updating the MAC values when an access is done incurs minimal overhead. This is a common strategy with tree-based ORAMs, which we will make integral use of in our scheme. We will also include client \( u_i \)'s counter \( \chi_u \) in the root MAC as before, to prevent rollback attacks (see below).

**Challenge**  Looking at Path ORAM, there exist several challenges when trying to add multi-client capabilities with our previous strategy. First, if we separate it into \( \phi \) separate ORAMs (which we will do), we actually end up with a very large blowup because of the recursion. At the top level, we will have \( \phi \) ORAMs, but each of those will have to have \( \phi \) ORAMs in turn to support the map, each of which will have \( \phi \) more, going down \( \log n \) levels. The overall complexity would be \( \phi^\log N \in \Omega(N) \). Additionally, the fact that Add cannot be performed without ReadAndRemove means that we cannot easily split the access into critical and non-critical parts like before.

**Rationale**  To remedy these problems, we institute the following major changes to Path ORAM:

1. **Unified Tagging:** Instead of separately tagging every block in each of the ORAMs, we will have a unified tagging system where a block \( x \) has the “same” tag in each of the separate client ORAMs. This allows us to avoid a branching factor for the recursive map. For a block \( x \), the map will resolve to a tag value \( t \) which describes its path in each one of the client ORAMs. Let \( h \) be a PRF mapping from \( [0, 2^\lambda) \times [1, \phi] \) to \( [0, N) \). The leaf that block \( x \) is percolating to differs for every ORAM and is pseudo-randomly determined by value \( h(t, i) \).

2. **Secure Block Removal:** The central problem with ReadAndRemove is that it is required before every Add so that the tree will not fill up with old, obsolete blocks which cannot be removed. Unlike the square-root ORAM, the shuffling process (eviction) happens locally and cannot know about other versions of a block which exist on different paths. We solve this problem by including metadata on each bucket. For every node in the tree, we include an encrypted array which indicates the ID of every block in that node. Removing a block from the tree can then be performed by simply changing the metadata to indicate that the slot is empty. It will be overwritten by the eviction routine with a real block if that slot is ever needed. If \( B \) is large, this metadata is substantially smaller than the real blocks. We can then store it in a less efficient classical ORAM described above which is itself multi-client secure. This allows us to take advantage of the better complexity provided by tree-based ORAMs for the majority of the data, while falling back on a simpler ORAM for the metadata which is independent of \( B \).
Input: Security parameter $\lambda$, number of blocks in each ORAM $N$, block size $B$, number of clients $\phi$, initialization sub-routine for multi-client secure classical ORAM $\text{MInit}$

Output: Initial ORAM state $\Sigma_{\text{init}}$, initial per client states $\{\text{st}_{u_1}, \ldots, \text{st}_{u_\phi}\}$

$\kappa \leftarrow \{0, 1\}^\lambda$

for $j = 1 \text{ to } \phi$ do
  $i = 0$
  $N_0 = N$
  while $n_i > 1$ do
    Initialize a tree $T_{j,i}$ with $N_i$ leaf nodes;
    Set eviction counter $e_{j,i} = 0$
    // The stash must also be stored on the server
    Create array $S_{j,i}$ with $Y$ blocks;
    // Use a sub-ORAM to hold block metadata
    $M_{j,i} = \text{MInit}(\lambda, 2n_i \cdot Z, Z \cdot \log n_i, \phi)$;
    $N_{i+1} = n_i \cdot \lceil \log n_i/B \rceil$;
    $i = i + 1$
  end
  Create a root block $R_j$;
  Set ORAM counter $\chi_j = 0$
  $\text{ORAM}_j = \text{Enc}(T_{j,0}, M_{j,0}, S_{j,0}, e_{j,0}) || \ldots || (T_{j,m}, M_{j,m}, S_{j,m}, e_{j,m}) || \chi_j || R_j$;
  Send $\text{st}_{u_j} = \{\kappa, \chi_j, e_{j,0}, \ldots, e_{j,m}\}$ to client $u_j$;
end
Send $\Sigma_{\text{init}} = \{\text{ORAM}_1, \ldots, \text{ORAM}_\phi\}$ to server;

Algorithm 16: $\text{Init}(\lambda, n, B, \phi)$, initialize ORAM

We also note that Path ORAM’s stash concept cannot be used in a multi-client setting. Since the clients do not have a way of communicating with each other out of band, all shared state (which includes the stash) must be stored in the RAM. This has already been noted by Goodrich et al. [35], and since the size of the stash does not exceed $\log N$, storing it in the RAM (encrypted and integrity protected) does not affect the overall complexity.

Similar to before, we also introduce an eviction counter $e$ for each ORAM. client $u_i$ will verify whether, for each of their recursive ORAMs, this eviction counter is fresh.

6.3.2 Details

Algorithm 16 initializes $\phi$ separate ORAMs and distributes the initial states (containing the shared key) to each client. These ORAMs $T_{j,i}$ each take the form of a series of trees. The first tree stores the data blocks, while the remaining trees recursively store the map which relates block addresses to leaf nodes. In addition to this, as described above, each tree has its own sub-ORAM to keep track of block metadata. The stash of each (sub-)ORAM is called $S_{0,i}$, and the metadata (classical) ORAM $M_{j,i}$.

To avoid confusion between different ORAM initialization functions, $\text{MInit}$ is a reference to Algorithm 14, i.e., initialization of a multi-client secure classical ORAM.
Input: Address \( x \), client \( u_i, st_{u_i} = \{ \kappa, \chi_i \} \), access sub-routine for multi-client secure classical ORAM

Output: The value of block \( x \)

// Let \( m \) be the depth of recursion, \( n_j \) be the number of blocks in tree \( j \)

Retrieve root block \( R \):
// Find tag \( t_m \) where \( x \) is mapped to
\[
pos = \frac{x}{n_j}; \; x_m = \left\lfloor \frac{\text{pos} \cdot (B/\lambda)}{\lambda} \right\rfloor; \; t_m = R[x_m];
\]
// Compute new tag \( t'_m \) for \( x \)
\[
t'_m \triangleq \{ 0, 2^\lambda \};
\]
for \( j = m \) to 0 do
\[
\text{leaves}_j = h(t_j, u_i) // Compute leaf of client \( u_i \)'s ORAM;
\]
Read path \( P(\text{leaves}_j) \) and \( S_{i,j} \) from \( T_{i,j} \), locating block \( x_j \);
Retrieve MAC values for \( P(\text{leaves}_j) \) as \( V \) and the stored counter as \( \chi'_i \);
if \( V \neq \text{MACPath}(\Sigma, st_{u_i}, P(\text{leaves}_j), S, \chi'_i) \) \( \vee \chi'_i \neq \chi_i \) then
\[
\text{Abort};
\]
Re-encrypt and write back \( P(\text{leaves}_j) \) and \( S_{i,j} \) to \( T_{i,j} \);
// Let \((a, b)\) be the node and slot that \( x_j \) was found at
MAccess(\( M_f \), (write, \( a \cdot Z + b \cdot \perp \)), \( u_i \));
if \( j \neq 0 \) then
\[
\text{leaves}'_j \triangleq \{ 0, 2^\lambda \} // Sample a new value for \( t \);
// Block \( x_j \) contains multiple \( t \) values
Extract \( t_{j-1} \) from block \( x_j \);
Update block \( x_j \) with new value \( t_{j-1}' \) and new leaf tag \( t_j' \);
else
Set \( v \) to the value of block \( x_j \);
If OP is a write, update \( x_j \) with new value;
end
Insert block \( x_j \) into the stash \( S_{i,j} \);
\[
\chi_i = \chi_i + 1;
\]
Update MAC of stash to \( \text{MAC}_s(S_{i,j}, \text{MAC} \text{of root bucket}, \chi_i, e_{i,j}) \);
// Update the block in other client’s ORAMs
for \( p \neq i \) do
\[
\text{Retrieve path } P(h(t, u_p)) \text{ from } T_{p,j} \text{ and update metadata so block } x_j \text{ is removed};
\]
Insert block \( x_j \) into the stash \( S_{p,j} \) of \( T_{p,j} \);
Update MAC of root bucket in \( T_{p,j} \);
end
output \( (v, st_{u_i} = \{ \kappa, \chi_i, e_{i,0}, \ldots, e_{i,m} \}) \);
end

Algorithm 17: Access(\( \text{OP}, \Sigma, st_{u_i} \)), perform Read or Write

For simplicity, we assume that \( \text{Enc}_\kappa \) encrypts each node of a tree separately, therewith allowing individual node access. Also, we assume authenticated encryption, using the per node integrity protection previously mentioned.

As noted above, the functions \( \text{ReadAndRemove, Add} \) can be used to implement \( \text{Access, Read, Write} \), which in turn can implement a simple interface \( \text{Access} \). Because our construction introduces dependencies between \( \text{ReadAndRemove} \) and \( \text{Add} \), in Algorithm 17 we illustrate a unified Access function for our scheme. The client starts with the root block and traverses the recursive map upwards to find the address of block \( x \) and finally retrieve it from the main tree. For each recursive tree, it retrieves a value \( t \) which allows it to locate the correct block in the next tree.
Input: Address $x$, new value $v$, client $u_i$, $st_{u_i} = \{k, \chi_i\}$
Output: The value of block $x$

for $j = 1$ to $\phi$
  for $r = 1$ to $m$
    Retrieve eviction counter $e_{j,r}$ for $T_{j,r}$;
    Retrieve path $\mathcal{P}(e_{j,r})$, $S_{j,r}$ and MAC chain $V$;
    // Verify integrity of the path and eviction counter
    if $V \neq \text{MACPath}(\Sigma, st_{u_i}, \mathcal{P}(\text{leaf}_j), S_{j,r}, \chi'_i, e_{j,r})$ then
      Abort;
    end
  end
  Read metadata for path from $M_{j,r}$;
  Move blocks out of the stash and down the path as far as possible;
  Reencrypt $\mathcal{P}(e_{j,r})$ and $S_{j,r}$ and write back to server;
  Update metadata for path $M_{j,r}$;
  $e_{j,r} = e_{j,r} + 1$;
end

Algorithm 18: Evict($\Sigma, st_{u_i}$) – Perform Evict

Input: $\Sigma$, $st_{u_i}$, path $\mathcal{P}$, stash $S$, $\chi$, eviction counter $e$
Output: Updated MAC values

for $j = \log n$ to $1$
  $V[j] = \text{MAC}_\chi(\text{contents of bucket } \mathcal{P}[j], \text{MAC of left child}, \text{MAC of right child});$
end

// Root MAC over the stash and tree parameters $\chi$ and $e$
$V[0] = \text{MAC}_\chi(S, \text{MAC of root bucket}, \chi, e);$ return $V$

Algorithm 19: MACPath($\Sigma, st_{u_i}, \text{path } \mathcal{P}, \text{stash } S, \chi, \text{eviction counter } e$)

After retrieving a block in each tree, the client marks that block as free in the metadata ORAM so that it can be overwritten during a future eviction. This is necessary to maintain the integrity of the tree and ensure that it does not overflow. At the same time, the client also marks that block free in the metadata of each other client and inserts the new block value into the root of their trees. This is analogous to the previous scheme where a client reads from their own ORAM and writes back to the ORAMs of the other clients.

Again, we avoid confusion between different ORAM access operations by referring to the multi-client secure classical ORAM access operation of Algorithm 15 as $\text{MAccess}$.

Algorithm 18 illustrates the eviction procedure. Since eviction does not take as input any client access, it is non-critical. The client simply downloads a path in the tree which is specified by eviction counter $e$ and retrieves it in its entirety. The only modification that we make from the original Path ORAM scheme is that we read block metadata from the sub-ORAM that indicates which blocks in the path are free and can be overwritten by new blocks being pushed down the tree.
6.3.3 Security Analysis

We start the security analysis by showing that, due to the MACs authenticating each data structure, a specific client \( u_i \) will read the same tag \( t \) from a tree in their ORAM with probability negligible in \( \lambda \).

**Lemma 6.7.** Let \( u_i \) be a client \( i \). For any two accesses to a map tree \( T_{i,j} \), \( 1 \leq j \leq m_i \), by client \( u_i \), which do not result in \( u_i \) aborting, the probability that they both return the same value \( t \) is negligible in \( \lambda \).

**Proof.** We start with the root block. Client \( u_i \) replaces each value with a fresh \( t_0 \) in the range \([0, 2^\lambda)\) after each access. So, if the server is honest, \( u_i \) will read the same value in two separate accesses only with probability \( 2^{-\lambda} \). For the case of a malicious server, \( u_i \) also keeps a counter \( \chi_i \) which is incremented after every access. The root block on the server additionally stores this counter along with a MAC that authenticates the block-counter combination. As long as the MAC is unforgeable with chance \( 1 - 2^{-\lambda} \), the probability that \( u_i \) does not abort on a bad block-counter combination is negligible.

After the root block, we continue with the map trees. The client will read a path in each tree which contains the target block, and next value \( t_j \). If the server is honest, \( u_i \) would have changed \( t_j \) since the last time it was accessed and the probability would again be \( 2^{-\lambda} \). Client \( u_i \) also has a MAC chain here tied to a counter which can be verified, so against a malicious adversary the probability is still negligible in \( \lambda \). \( \square \)

With that lemma, we can prove that our construction is secure based on the fact that the \( t \) values induce a uniform distribution of blocks across the leaf nodes and that no client will have a collision in their \( t \) values with any non-negligible probability.

**Theorem 6.8.** Our tree-based construction \((\text{Init}, \text{Access})\) is a multi-client Oblivious RAM secure against malicious adversaries.

**Proof.** If \( h \) is a PRF, then assigning leaf nodes to blocks as \( h(t, i) \) for client \( u_i \) will result in a (pseudo-)random distribution over the leaf nodes for every block in every tree. By Lemma 6.7, even against a malicious adversary, with all but negligible probability no client will make two accesses that return the same value \( t_i \). By induction, this means that the paths read in each tree when a client accesses their own ORAM will be distributed pseudorandomly, independent of the virtual block being accessed. Thus, a client reading from their own ORAM cannot leak any information that would allow an adversary to distinguish between two access patterns.
When clients write to other clients’ ORAMs, they directly and deterministically access the stash. The clients additionally read and write with the sub-ORAM, which is in itself multi-client secure. Since they always execute the same number of accesses \((\log n)\) per tree on this ORAM, and the number of accesses is the only thing leaked to the adversary with a secure ORAM. This information cannot give an advantage to the adversary in distinguishing access patterns.

The last algorithm is eviction. Since the path chosen during eviction is deterministic (based on the counter) and independent of any accesses done by any client, it is straightforward to see that it also will induce a pattern on the server which is indistinguishable.

### 6.3.4 Complexity

The complexity of our scheme is dominated by the cost of an eviction. For a client to read a path in each of \(O(\log N)\) recursive trees, for each of the \(\phi\) different ORAMs, it takes \(O(\phi \cdot B \cdot \log^2 N)\) communication. Additionally, the client must make \(O(\phi \cdot \log^2 N)\) accesses to a metadata ORAM. If \(\mu(N, B)\) denotes the cost of a single access in such a sub-ORAM, the overall complexity is then \(O(\phi \cdot \log^2 N \cdot (B + \mu(N, \log N)))\). Taking the hierarchical ORAM as a sub-ORAM, the total worst-case communication complexity computes to \(O(\phi \cdot \log^2 N[B + \log^4 N])\). If \(B \in \Omega(\log^4 N)\) then the communication complexity, in terms of blocks, is \(O(\log^2 N)\), otherwise it is at most \(O(\log^5 N)\), i.e., with the assumption \(B \in \Omega(\log N)\) (minimal possible block size).

Although a complexity linear in \(\phi\) may seem at first to be expensive, we stress that this is a substantial improvement over naive solutions which achieve the same level of security. The only straightforward way to have multi-client security against malicious servers is for each client to append their updates to a master list, and for clients to scan this list to find the most updated version of a block during reads. This is not only linear in the size of the database, but in the number of operations performed over the entire life of the ORAM.

One notable difference in parameters from basic Path ORAM is that we require a block size of at least \(c \cdot \lambda\), where \(c \geq 2\). Path ORAM only needs \(c \cdot \log n\), and for security parameter \(\lambda\), \(\lambda > \log N\) holds. In our scheme, the map trees do not directly hold addresses, but \(t\) values which are of size \(\lambda\). In order for the map recursion to terminate in \(O(\log N)\) steps, blocks must be big enough to hold at least two \(t\) values of size \(\lambda\). If the block size is \(\Omega(\lambda^2)\), we can also take advantage of the asymmetric block optimization from Stefanov et al. [7] to reduce the complexity to \(O(\phi \cdot (\log^6 n + B \cdot \log N))\). Then, if additionally \(B \in \Omega(\log^5 N)\), the total complexity is reduced to \(O(\log N)\) per client.
6.3.5 Conclusion

We have presented the first techniques that allow multi-client ORAM, secure even in the face of fully malicious servers. Our multi-client ORAMs are reasonably efficient with complexities between $O(\log N)$ to $O(\log^5 N)$ per client, depending on the underlying block size. Future work will focus on efficiency improvements, e.g., reducing worst-case complexity to being sub-linear in $\phi$. Additionally, the question of whether tree-based constructions are more efficient than classical ones is not as clear in the multi-client setting as it is for a single client. Although tree ORAMs are more efficient for a number of parameter choices, they incur substantial overhead from using a sub-ORAM to hold tree metadata. This is not required for the classical constructions. Future research may focus on achieving a “pure” tree-based construction which does not depend on another ORAM. Finally, it may be interesting to investigate whether multiple clients can be supported with a more fine-grained access control. For example, instead of every client have full permissions to the ORAM, can they have separate keys and somehow share only pieces of their individual databases.
Bibliography


Bibliography


[40] 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 with integrity verification. 2014.


