Data-Oblivious Algorithms for Privacy-Preserving Access to Cloud Storage

by

Olga Ohrimenko

A dissertation submitted in partial fulfillment of the

requirements for the degree of Doctor of Philosophy

in The Department of Computer Science at Brown University

PROVIDENCE, RHODE ISLAND

May 2014

This dissertation by Olga Ohrimenko is accepted in its present form
by The Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date________________ ________________________________

Roberto Tamassia, Ph.D., Advisor

Recommended to the Graduate Council

Date________________ ________________________________

Michael T. Goodrich, Ph.D., Reader

Date________________ ________________________________

Anna Lysyanskaya, Ph.D., Reader

Approved by the Graduate Council

Date________________ ________________________________

Peter M. Weber, Dean of the Graduate School

Acknowledgements

This thesis would never have been possible without my Ph.D. advisor, Roberto Tamassia. Roberto has provided me with the best guidance I could ever hope for from an advisor. Besides the technical knowledge he shared with me over the wonderful three years that I have worked with him, he has been an inspiring mentor giving me advice on ethics in academia and research. Roberto, I especially enjoyed the white board meetings we had and the rushed spirit of deadlines. Thank you for your insightful comments on all my writings and presentations, and for always finding time to meet and talk to me. It has been an honor to be your student!

I would like to express gratitude to my wonderful thesis committee members, Michael Goodrich and Anna Lysyanskaya. I had the pleasure to collaborate with Michael from the very first days of this thesis. During this time he always shared not only his many ideas on oblivious algorithms but also his research enthusiasm and work ethics. Anna Lysyanskaya always found time to give me feedback on my presentations, academic advice and helped me to improve the cryptography side of several sections of this thesis.

Eli Upfal has indirectly mentored and overseen my Ph.D. over all five years I have been at Brown. Eli, I enjoyed meetings and courses that I have either taken or TAed with you. Every interaction has always been fun and I learnt something from you every time. I will greatly miss running into you in the halls of the CIT. Finally, I am especially pleased to have had a chance to collaborate with you towards the end of my Ph.D.

Christian Cachin, Seny Kamara, Claire Mathieu, Michael Mitzenmacher, Babis Papamanthou and Nikos Triandopoulos have been amazing collaborators and many projects I was involved in would not have been possible without their involvement. Thanks to Roberto, I also had an opportunity to work with and learn from wonderful Brown students including John Boreiko, Josh Brown, Esha Ghosh, Duy Nguyen and Hobart Reynolds.

Life inside and outside of the walls of the CIT was made fun with the help of Andrew, Aparna, Arjun, Carleton, Emanuel, Fabio, Justin, Layla, Matteo, Serdar, Silvia and Yuri. I have also been lucky to have Duana, Linda and Ulya and their remote support. Special thanks to my TG for always being there.

Finally, I would like to thank my family for undertaking an unpredictable journey to Australia which eventually allowed me to pursue a research career which may not have been possible otherwise.

This thesis was supported in part by the National Science Foundation under grants CCF–0830149, CNS–1012060, CNS–1228485, IIS–1212508 and OCI–072480, and a NetApp Faculty Fellowship.

I dedicate this thesis to my grandmothers, Maria and Zoya, who are no longer with us.

Abstract of “ Data-Oblivious Algorithms for Privacy-Preserving Access to Cloud Storage ” by Olga Ohrimenko, Ph.D., Brown University, May 2014

Cloud storage has emerged as the next generation of data storage where users can remotely store their data and leave its management to a third party, e.g., Amazon S3, Google Drive or Microsoft Azure. However, the fact that users no longer have physical possession of their data raises new challenges in terms of data privacy. Storing the data in encrypted form is a key component in maintaining privacy against the storage provider. However, encryption alone is not enough since information may be leaked through the pattern in which users access the data. In this thesis, we describe algorithms that allow *data-oblivious* access to remotely stored data. That is, access patterns of such algorithms depend only on the size of the outsourced data and algorithm input, but not their content. Hence, such algorithms reveal nothing about the data they are processing.

We start by describing a general method that obliviously simulates user requests to outsourced data of size n and adds $O(\log n)$ overhead in the average case, succeeding with very high probability. This method assumes a private workspace of size $O(n^\epsilon)$ on the user side, for any given fixed positive constant ϵ, and does not maintain a state between data requests. We then show how to deamortize our method to achieve $O(\log n)$ overhead in the worst case. Our deamortization technique is general and can be applied to several existing oblivious simulations.

The next oblivious simulation technique presented in this thesis improves over the $O(\log n)$ solution and demonstrates an interplay between system parameters such as latency, bandwidth, and the size of the user’s private memory. We show that if a user exchanges messages of size $O(n^{1/c} \log n)$ with the storage provider, for some constant $c \geq 2$, and has access to private memory of the same size, then our method can achieve $O(1)$ access overhead in the worst case with very high probability.

Finally, we study application-specific access patterns and look at how to make them oblivious without using the general oblivious simulation methods mentioned above. In particular, we show how one can access and perform a computation in an oblivious fashion on an externally stored graph. We also show that a number of classic graph drawing algorithms can be efficiently implemented in this framework while maintaining user privacy.

Contents

List of Tables

List of Figures

Chapter One

Introduction

Outsourcing data to a remote storage provider like Amazon S3, Dropbox, Google Drive or Microsoft Azure has become a common alternative to data management for single users and companies. For example, in 2012 Amazon alone reported storing one trillion objects in their storage service Amazon S3 [4]. The popularity of storing data in the "cloud" is explained by its cost effectiveness, since clients pay only for the computational resources that they use, and reliability, a result of redundancy and resilience of multiple servers around the globe. However, the lack of physical possession of the data creates new challenges in ensuring privacy of the outsourced data since storage providers may have commercial interest in their customers' data or inadvertently a third party may gain access to a user's data. These concerns are not unfounded: for example in 2011 an error in Dropbox code led to temporary allowing unauthenticated access to their users' data [14]

A key component for users to maintain the privacy of their data is to store the data in encrypted form using a key known only to the user. However, simply encrypting the data is not sufficient to achieve privacy since information about the data and the user may be leaked by the pattern in which the user accesses it. Consider the following example. A user outsources three encrypted records x, y and z. She then wishes to execute the following program:

```
read $x$
if $x \geq 100$ then read $y$
else read $z$
```

Although the value of x is encrypted, accessing the encrypted values of x and y reveals that x is greater or equal to 100. Similarly, accessing x and then z reveals that $x < 100$.

In another example consider access pattern to a collection of documents outsourced by a company where documents can be accessed by the company's employees, i.e., a multi-user scenario. By observing which documents the users access and comparing their access patterns, the storage provider can infer access control on the documents within the company, as well as the collaboration network of the employees. Again notice that such information can be inferred even if all documents are encrypted.

Thesis Objective In this thesis, we are interested in designing data structures and algorithms that protect user privacy while storing and accessing data at the cloud storage provider. The information we are trying to protect is the content of the data and the pattern of access to the data. The latter includes hiding such information as frequency of accesses to individual data elements, whether the access is a read or a write, if the currently accessed item has been accessed before and when, and finally any dependencies between accesses (e.g, in the above example y is accessed when $x \geq 100$). Storing data in an encrypted form is a key component for hiding the content of the data. However, one has to design other solutions to hide the access pattern. The access pattern depends on the application or the algorithm the customer of cloud storage uses to access her data. For example, if the user always reads all of her data and never updates any records, then a scan over encrypted data does not reveal anything to the storage provider other than that a request was made. However, only a minority of applications require accessing the whole data collection.

The pattern in which the user accesses her data may depend on the application the user is running, its input and outsourced data the user retrieves. If one knows a priori how the data is accessed, then one can design algorithms that make accesses without revealing the originally intended access locations. We refer to such algorithms as *data-oblivious algorithms*, since anyone observing data locations accessed by these algorithms, besides the user herself, is oblivious to the data that the algorithms are operating on and what data is being requested. For example, Batcher's sorting network [5] is a data-oblivious algorithm for sorting an array of elements. In other words, this algorithm sorts the array by accessing its elements independently of the underlying values. Moreover, this algorithm's memory accesses are deterministic and depend only on the size of the array but not its values.

However, there are also cases when data accesses are not as predictable as, for example, in insertion sort. Therefore, we are interested in designing solutions that make every request to the data look independent of the elements being requested now or before. We refer to the schemes that take any sequence of requests and simulate them obliviously as *oblivious simulation schemes*.

Oblivious RAM Goldreich and Ostrovsky [21, 18] posed a problem similar to ours in the context of protecting software from illegitimate duplication and consequent redistribution. In particular, they consider inferences that can be made by an adversary observing interactions of the trusted CPU running the program with untrusted memory, i.e., a sequence of addresses accessed. They define *oblivious RAM* (ORAM) as an interface between the CPU and the memory that hides the access pattern by accessing the memory in an oblivious manner. That is, the probability distribution of the sequence of (memory) addresses accessed during an execution depends only on the input length and is independent of the particular input. Goldreich and Ostrovsky [21, 18] presented two oblivious RAM constructions with different performance guarantees, that we refer to as square root and hierarchical solutions.

Protecting the access pattern of a program using local memory versus accessing data from the "cloud" is similar from a theoretical point of view, however, several aspects of the problem change when one switches to the cloud setting. First, in the remote storage case, the user is a computer or a mobile device that has access to private memory. The size of the private memory is likely to be proportional to the size of the data that the user has outsourced, e.g., big documents are less likely to be accessed from mobile devices. Secondly, the number of roundtrips one has to make to the cloud provider to access the data becomes a bottleneck when accessing data that is stored in a different state or country. Finally, the size of the packets exchanged between the user and the cloud provider is no longer a single element and the cloud API supports a richer set of queries than RAM.

Desired Properties We are interested in designing solutions that are efficient and provably secure. For example, our schemes should not defeat the purpose of cloud computing by using private memory equal in size to the outsourced data. It is also desirable to minimize any overhead the oblivious scheme adds compared to the non-oblivious implementation of the same application. More specifically, if the user outsources n data records, then we wish to simulate a sequence of requests by adding a constant or sublinear in n access overhead. Cloud storage providers charge customers for storing the data as well as accessing it. Hence, oblivious schemes should also take into account the monetary cost they add. Oblivious schemes should also adhere to common security definitions and, hence, protect access privacy for any request sequence, even if the adversary can influence what this sequence may be. Informally, we wish to design schemes that make accesses to remote data storage depending only on publicly known parameters, such as the size of the outsourced data, the size of the user's private memory, the number of requests in the original sequence, and the number of elements the user can exchange with the server. The access pattern should not depend on the data outsourced nor the application that the user is running.

We view the storage server as an *honest-but-curious* adversary, who correctly performs the storage and retrieval operations requested by the user, but is nevertheless interested in learning as much from her data as possible (indeed, some cloud computing companies are basing their business model on this goal). We do not consider timing side channel attacks that measure the time it takes for the user to make a request or process the received data. Hiding the length of the request sequence is also outside the scope of this thesis.

Thesis Contributions This thesis addresses several aspects of the problem of hiding access patterns. To this end, we incorporate multiple cryptographic primitives and various probabilistic data structures in order to build efficient and secure solutions. In particular, we develop two oblivious access schemes: the $\log n$-hierarchical solution and the constant oblivious storage solution. Assume the outsourced data set consists of n elements. The former scheme improves over the known bound on the access overhead by adding $O(\log n)$ accesses for the oblivious simulation by introducing the idea of a common stash shared between multiple cuckoo hash tables. The latter solution exploits a tradeoff between network bandwidth and user private memory to achieve a constant access overhead. This solution introduces a recursive oblivious storage scheme that uses a novel algorithm for constructing a permutation obliviously. We then observe that many oblivious schemes strive to minimize the amortized access overhead while leaving the worst case to take $\Omega(n)$ accesses, which may not be desirable in certain applications. We develop a deamortization technique for the $\log n$-hierarchical solution and the square root solution of [18]. Finally we consider the access pattern of graph drawing algorithms and study their data-oblivious variants by performing scanning and computation over the Euler tour representation of a tree.

Thesis Outline The rest of this thesis proceeds as follows. In the next chapter, an overview of cryptographic primitives and data structures used in this thesis is presented. We also give the security definition of a data-oblivious algorithm and present the square root solution from [18, 21] as a warm-up for our oblivious schemes.

In the outline of the remaining chapters, we denote with n the size of the outsourced data set.

In Chapter 4, we present our first oblivious simulation scheme. This scheme is based on a hierarchy of $O(\log n)$ hash tables, where the top hash tables are implemented as hash tables with buckets followed by cuckoo hash tables that share a common stash. This scheme incurs $O(\log n)$ access overhead since the simulation must access every level. We also present experimental results to show that the overhead of the scheme is small in practice. A preliminary version of this chapter appeared in [27].

The second oblivious simulation scheme is presented in Chapter 5. This scheme achieves constant overhead if private memory of size $O(n^{1/c} \log n)$, for any constant $c \geq 2$, is available to the client and if the client and the cloud storage provider can exchange messages consisting of $O(n^{1/c} \log n)$ elements. This chapter is based on the work presented in [26] and [40, 41]. An extension of the above construction that uses memory of size $O(n^{1/c})$ is presented in [40, 41].

In Chapter 6, we present deamortized variants of the original square root solution, the $\log n$ hierarchical solution from Chapter 4 and constructions from Chapter 5.

This technique shows that by doubling the space overhead at the cloud provider, one can achieve $O(\log n)$ access overhead in the worst-case, avoiding the occasional $O(n)$ downtime that was present in the original exposition of these methods. A preliminary version of this chapter appeared in [25].

We study several application-specific data-oblivious algorithms in Chapter 7. We consider the problem of drawing graphs that are stored remotely. Such algorithms involve traversing and performing computations on trees and graphs. We describe a computational model for this problem that we call compressed-scanning. Several graph drawing algorithms are adapted to fit the compressed-scanning model and shown to be data-oblivious. A preliminary version of this chapter appeared in [28].

Finally, we review the work related to oblivious RAM and more recent advances on oblivious cloud storage (OS) in Chapter 2. We also consider relevant concepts such as private information retrieval and oblivious transfer and show how they differ from oblivious RAM. We conclude the thesis and describe directions for future work in Chapter 8.

Chapter Two

Related Work

In this chapter we review work related to oblivious RAM, and its extension, oblivious storage, that supports a richer set of queries than the RAM model. Table 2.1 provides the comparison of different aspects of ORAM simulations. We also review oblivious storage schemes in a two-cloud model where storage providers are non-colluding. We end the chapter with the description of several applications of ORAM where privacy of the access pattern is vital for the security of these applications.

2.1 Oblivious RAM

Prior work on oblivious RAM addresses the trade-off between the size of the client's memory, the size of the messages exchanged between the client and the server in a single roundtrip, the access overhead, and the space overhead at the storage provider, i.e., the additional space used beyond the n items.

Based on the assumptions about the client, oblivious RAM models can be classified into *stateless* and *stateful* solutions. A stateless oblivious RAM is not allowed to keep a state between requests and hence can be used in a multi-user scenario. Stateful solutions assume that the user Alice keeps information in a private storage (which she maintains), which helps her perform her accesses obliviously in the remote storage.

Stateless oblivious RAM simulation was first proposed by Goldreich and Ostrovsky in [21], who present a preliminary simple solution with $O(\sqrt{n}\log^2 n)$ amortized access overhead, referred to as the *square-root solution*, and a more complex solution with $O(\log^3 n)$ amortized access overhead. Goodrich and Mitzenmacher [24] improve this result by giving a method with $O(\log^2 n)$ amortized access overhead with high probability. Recently Kushilevitz *et al.* [34] show that techniques from [24] can be extended to obtain $O(\log^2 n/\log\log n)$ amortized access overhead. All the above stateless methods utilize a private memory of size only $O(1)$ for Alice, an overly restrictive assumption in practice.

Other solutions [51, 53] improve the overall access overhead by assuming that a client has a workspace of non-constant size. Williams and Sion [51] achieve $O(\log^2 n)$ expected amortized access overhead and $O(n \log n)$ space overhead with $O(\sqrt{n})$ private memory. Williams *et al.* [53] improve the method from [51] to achieve $O(\log n \log\log n)$ amortized access overhead.

Damgård *et al.* [11] and Ajtai [1] present stateless oblivious RAM simulations that do not access a random oracle nor make cryptographic assumptions about the existence of pseudorandom functions. Using different techniques Damgård *et al.* [11] and Ajtai [1] show that amortized access overhead of $O(\log^3 n)$ is possible for oblivious RAM simulation without using random functions. The authors of [11] also show that a minimum of $O(\log n)$ random bits are required for oblivious request simulation for a memory of size n.

2.2 Oblivious Cloud Storage

Boneh *et al.* [7] deviated from standard ORAM schemes by introducing the *oblivious storage* (OS) problem that, as they argue, is more realistic and natural than the ORAM simulation problem. They study methods that separate access overheads and the overheads needed for rebuilding the data structures on the server, providing, for example, $O(1)$ amortized overhead for accesses with $O(\sqrt{n \log n})$ overhead for rebuilding operations, assuming a similar bound for the message size and the size of the private memory on the client.

Stefanov *et al.* [47] study the oblivious storage simulation problem from a practical point of view, with the goal of reducing the worst-case bounds for data accesses. They show that one can achieve an amortized overhead of $O(\log n)$ and worst-case performance $O(\sqrt{n})$, with $O(\epsilon n)$ storage on the client, for a constant $0 < \epsilon < 1$, and an amortized overhead of $O(\log n)$ and similar worst-case performance, with a client-side storage of $O(\sqrt{n})$. The scheme of [44] provides a tree-based construction that uses $\tilde{O}(n \log n)$ server storage and incurs $\tilde{O}(\log^2 n)$ overhead on each access when the client has access to $O(\epsilon n)$ private memory (the $\tilde{O}$ notation hides the $\mathsf{poly} \log \log n$ terms). Both schemes [47] and [44] can be modified using a recursive ORAM construction to require $O(\sqrt{n})$ and $O(1)$ private memory, correspondingly, and instead incurring a logarithmic factor in their access overhead.

Path-ORAM is a simple oblivious construction proposed in [48] that makes $O(\log n)$ accesses to the server and uses a position map and a stash both stored at the client. The size of the position map is n but can be reduced by applying the recursive ORAM construction as in [47] and [44], while the size of the stash is logarithmic in n. With recursive construction, the method achieves $O(\log^2 n)$ access overhead and $O(\log n)$ client storage.

An interesting construction by Williams and Sion [52] achieves a constant overhead by using computation power of the server. Most oblivious constructions incur roundtrip overhead since they do not know where the item is in the construction (the exceptions are [47, 44, 48] where a position map is maintained). Moreover, they also cannot reveal to the server where the item was found since this reveals when items were updated last. Williams and Sion outsource the search procedure to the server by encrypting all possible search paths for the item that one can take in the construction. The server then executes this search by working only on encrypted data and, hence, remains oblivious to where he finds the item. However, occasional reshuffle operation required by the solution adds a $\tilde{O}(\log^2 n)$ amortized overhead.

The oblivious simulations described above consider a single-client scenario where all accesses, including read-only accesses, are processed sequentially. Extending these solutions to support parallel access is not trivial since they guarantee obliviousness by sequentially maintaining a state. The works of Stefanov and Shi [46] and Williams *et al.* [54] allow parallel access. The clients access oblivious storage of [46] via a load balancer that is responsible for scheduling client requests. On the other hand, the clients of [54] first access a log file of pending requests and then determine if they should send a request for a real or a fake item.

2.3 Multicloud Model

Lu and Ostrovsky [35] and Stefanov and Elaine Shi [15] consider the model of outsourcing the data to two non-colluding storage providers, e.g, Amazon S3 and Microsoft Azure. The construction of [35] splits a hierarchical ORAM construction between the servers such that the servers store alternating levels only. The simulation proceeds as it does in the single-server case with accesses also alternating between the servers. However, when it is time to rebuild one of the levels at, the user uploads the data from one of the servers to another one according to a new pseudorandom function. This way the servers cannot correlate the accesses before rebuild and after since one server sees accesses to items before the rebuild and not after, while another one sees only accesses after the rebuild. This construction achieves

$O(\log n)$ amortized access overhead by avoiding an expensive shuffle sort by simply uploading items from one server to another one. The construction of [15] uses [47] as the underlying ORAM construction at each server. The interaction between the servers is done directly, not via the client as it is in [35]. Hence, the client does not participate in the shuffling phase. The scheme of [15] achieves a constant overhead. The authors also propose how the clients can verify if shuffling was correct when one of the servers is malicious.

2.4 ORAM Applications

Lu and Ostrovsky [36] use ORAMs for garbling RAM programs. In their application, a client outsources the execution of a RAM program to a server and wishes to hide what type of computation is being executed. An ORAM scheme, similar to the one of [52], is used to hide the access pattern of the garbled program. Gordon *et al.* [29] use ORAM of [44] to securely outsource a computation of a program that can be executed in sublinear time on a RAM. As an application, they consider a server that stores a large dataset D and a client who has a small input x. The client then wishes to perform a private query on D with x as the input, e.g., a search query.

Oblivious RAM simulation has also been used to protect against traffic analysis in a networked file system [55]. Cash *et al.* [8] use ORAM to create a dynamic version of proofs of retrievability scheme where the storage supports updates. Their scheme allows the user to perform efficient reads, writes to outsourced storage, and provides an audit functionality for verifying that the server maintains the latest copy of her data. Williams *et al.* [54] propose an ORAM construction for outsourced filesystem application where multiple users can access the storage in parallel.

Recently, several hardware ORAM implementations have been proposed [37, 17, 43]. The works of [37] and [17] design the architecture for the hardware, while [43] is a simulator for a secure processor. All three are based on the Path ORAM construction [48]. Optimizations and extensions for incorporating Path ORAM in a secure hardware design, including integrity verification, are described in [43].

Table 2.1: Comparison of Several Oblivious Simulation Methods where $c \geq 2$, $0 < \epsilon < 1$ and $0 < \tau < 1$. The $\tilde{O}$ notation hides $\mathsf{poly} \log \log n$ terms. The server storage with $*$ indicates temporary storage that is required during the rebuild phase while both solutions require only $O(n)$ during the access phase. Hence, deamortized versions of these constructions require $O(n \log n)$ storage during the access phase as well.

	User Memory	Message Size	Server Storage	Amortized Access Overhead	Worst-Case Access Overhead
Goldreich-Ostrovsky [21] $\sqrt{n}$	$O(1)$	$O(1)$	$O(n)$	$O(\sqrt{n}\log^2 n)$	$O(n \log^2 n)$
Goldreich-Ostrovsky [21] $\log n$	$O(1)$	$O(1)$	$O(n \log n)$	$O(\log^3 n)$	$O(n \log^3 n)$
Shi *et al.* [44] (BST Bucket)	$O(1)$	$O(1)$	$\tilde{O}(n \log n)$	$\tilde{O}(\log^2 n)$	$\tilde{O}(\log^3 n)$
Williams-Sion [52]	$O(\log n)$	$O(\log n)$	$O(n)$	$\tilde{O}(\log^2 n)$	$\tilde{O}(\log^2 n)$
Williams *et al.* [53]	$O(\sqrt{n})$	$O(1)$	$O(n)$	$O(\log^2 n)$	$\tilde{O}(n)$
Goodrich-Mitzenmacher [24]	$O(n^{1/\epsilon})$	$O(n^{1/\epsilon})$	$O(n)$	$O(\log^2 n)$	$O(n)$
Stefanov *et al.* [47]	$O(\sqrt{n})$	$O(1)$	$O(n)$	$O(\log^2 n)$	$O(\sqrt{n})$
Stefanov *et al.* [48]	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log^2 n)$	$O(\log^2 n)$
Kushilevitz *et al.* [34]	$O(1)$	$O(1)$	$O(n)$	$O(\log^2 n / \log\log n)$	$O(\log^2 n / \log\log n)$
Boneh *et al.* [7]	$O(\sqrt{n \log n})$	$O(\sqrt{n \log n})$	$O(n)$	$O(1)$	$O(n \log n)$
$\log n$-hierarchical (Chapter 4, [27])	$O(n^\epsilon)$	$O(n^\epsilon)$	$O(n)$	$O(\log n)$	$O(n)$
Deamortized $\sqrt{n}$ (Chapter 6, [25])	$O(1)$	$O(1)$	$O(n)$	$O(\sqrt{n}\log^2 n)$	$O(\sqrt{n}\log^2 n)$
Deamortized $\log n$-hierarchical (Chapter 6, [25])	$O(n^\tau)$	$O(n^\tau)$	$O(n)$	$O(\log n)$	$O(\log n)$
Constant overhead scheme (Chapters 5.4, 6, [40, 41])	$O(\sqrt{n}\log n)$	$O(\sqrt{n}\log n)$	$O(n \log n)^*$	$O(1)$	$O(1)$
Constant overhead scheme (Chapters 5.5, 6, [40, 41])	$O(\sqrt[c]{n}\log n)$	$O(\sqrt[c]{n}\log n)$	$O(n \log n)^*$	$O(c \log n)$	$O(c \log n)$

Chapter Three

Preliminaries

In this chapter we describe tools and security definitions we use for our oblivious constructions. We also give an overview of the square root oblivious RAM construction by Goldreich [18] as a warm-up for our solutions.

3.1 Notation

We refer to the user who wishes to store her data remotely with a storage provider Bob, as Alice. We use the terms user and client interchangeably. Alice has a dataset A of n elements, that we also may refer to as items. Each item consits of a pair (x, v), where $x \in \{0, \cdots, n-1\}$ is the key and v is the corresponding value. We sometimes refer to the storage provider as the server.

3.2 Adversarial Model

We consider a probabilistic computationally bounded adversary $\mathcal{A}$ who is limited to running in polynomial time in the size of the private keys and seeds used by the cryptographic primitives defined in Section 3.3. The adversary is *honest-but-curious* [20], in that he correctly performs all operations and does not tamper with the data. However, all of our solutions can protect data privacy in the *malicious* case as well, by simply extending the technique of Goldreich [21] and using message authentication codes [19].

In our scenario, the adversary $\mathcal{A}$ is either the storage provider, Bob, or anyone observing the interaction between Bob and user Alice.

3.3 Cryptographic Primitives

We let $k \in \mathrm{N}$ to denote the security parameter used in this paper. We now define the cryptographic tools we use in our constructions.

Negligible Function [19] A function $\nu : \mathbb{N} \to \mathbb{R}$ is called negligible if for every positive polynomial p there exists N such that for all $k > N$,

$$\nu(k) < \frac{1}{p(k)}$$

We will often say that undesirable events happen with negligible probability, that is with probability $1/2^k$, where k is the security parameter.

Computational Indistinguishability Two distributions D_1 and D_2 are computationally indistinguishable, if for every probabilistic polynomial-time (PPT) algorithm $\mathcal{A}$ there exists negligible function ν such that

$$|\Pr[\mathcal{A}(1^k, D_1) = 1] - \Pr[\mathcal{A}(1^k, D_2) = 1]| \leq \nu(k)$$

Here, input 1^k means that algorithm $\mathcal{A}$ runs in time polynomial in security parameter k.

Semantically Secure Encryption (Enc-IND-CPA) We use a *symmetric encryption scheme* $(\mathsf{Enc}_{\mathsf{key}}, \mathsf{Dec}_{\mathsf{key}})$ where $\mathsf{key} \leftarrow \{0,1\}^k$. We require this scheme to be secure against the chosen-ciphertext attack (CPA) for multiple messages [22, 19].

During this attack an adversary $\mathcal{A}$ is allowed to make queries to oracles $\mathsf{Enc}_{\mathsf{key}}$ and $\mathsf{Dec}_{\mathsf{key}}$ on a polynomial number of sequences of l messages of his choice. After this "warm-up" phase, $\mathcal{A}$ comes up with two sequences, m_0 and m_1, of l messages and gives them to a challenger. The challenger secretly picks a bit b and calls $\mathsf{Enc}_{\mathsf{key}}$ on each message of sequence m_b. Let C be the sequence of ciphertexts that correspond to m_b. The challenger gives C to $\mathcal{A}$ who continues querying $\mathsf{Enc}_{\mathsf{key}}$ and $\mathsf{Dec}_{\mathsf{key}}$ on any sequence of ciphertexts except those in C for a polynomial number of times. Finally, the adversary's task is to guess bit b. We call the above game *Enc-IND-CPA* and say that $\mathcal{A}$ wins the game if he correctly guesses b. Then, $(\mathsf{Enc}_{\mathsf{key}}, \mathsf{Dec}_{\mathsf{key}})$ is said to be secure if for all PPT adversaries, the probability of winning game Enc-IND-CPA is at most $1/2 + \mathsf{negl}(k)$. We omit using key when referring to $(\mathsf{Enc}, \mathsf{Dec})$. For an intuition behind Enc-IND-CPA secure encryption scheme, consider encrypting a message padded with a different random nonce each time it is encrypted. Hence, re-encryptions of the same plaintext look different with very high probability.

Pseudorandom Function (PRF) [31] Let $\mathcal{F}_{\mathsf{seed}} : \{0,1\}^m \to \{0,1\}^{m'}$ be a family of efficiently computable functions keyed using a seed from the set $\mathsf{Seeds}(\mathcal{F}) = \{0,1\}^k$. Then $\mathcal{F}_{\mathsf{seed}}$ is a *pseudorandom function* if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a negligible function ν such that:

$$|\Pr[\mathcal{A}^{\mathcal{F}_{\mathsf{seed}}(\cdot)}(1^k) = 1] - \Pr[\mathcal{A}^{F(\cdot)}(1^k) = 1]| \le \nu(n)$$

where F is chosen from the set of all functions mapping m-bit strings to m'-bit strings. For our purposes m will be at most m'.

Pseudorandom Permutation (PRP) [31] Let $\mathcal{R}_{\mathsf{seed}} : \{0,1\}^m \to \{0,1\}^m$ be a family of efficiently computable permutations keyed using a seed from the set

$\mathsf{Seeds}(\mathcal{R}) = \{0,1\}^k$. Then $\mathcal{R}_{\mathsf{seed}}$ is a *pseudorandom permutation* if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a negligible function ν such that:

$$|\Pr[\mathcal{A}^{\mathcal{R}_{\mathsf{seed}}(\cdot)}(1^k) = 1] - \Pr[\mathcal{A}^{R(\cdot)}(1^k) = 1]| \leq \nu(n)$$

where R is chosen from the set of all permutations on input of size m.

We will describe and prove security of our constructions assuming access to a truly random function F. However, in practice, we substitute access to F with an access to a pseudorandom function. Similarly, for pseudorandom permutations we will assume access to a truly random permutation.

Given an array A of n (key,value) pairs (x, v) where $x \in [1, n]$, we denote the permutation π of A as $B = \pi(A)$, where $\pi = \mathcal{R}_{\mathsf{seed}}$ and $B[x] = A[\pi(x)]$, $\forall x \in [1, n]$. We will use the same notation when A and B are encrypted. We refer to the original permutation of A, as permutation π_0. For A, sorted using x, π_0 is the identity.

3.4 Hash Tables

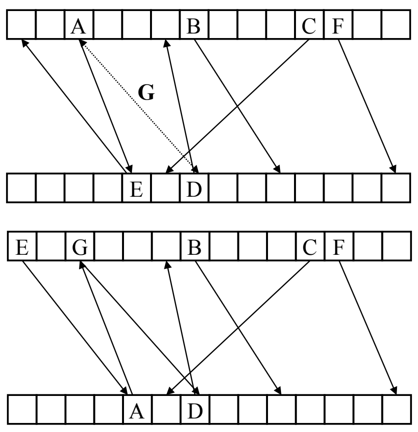
We now describe variations of hash tables that we use for the oblivious constructions presented in this thesis.

Hash Tables Hash tables with buckets remedy the collision problem of regular hash tables by allowing more than one item to be mapped to a location of a hash table, called bucket. Then, insertion of n items in a table with n buckets of size $\log n/\log\log n$ causes an overflow with probability $1/n$ [39]. However, our construction uses hash tables of sizes ranging between $O(\log n)$ and $O((\log n)^7)$ with buckets of size $c \log n / \log \log n$ to bound the probability of overflow to $1/n^c$ (see Section 4.3).

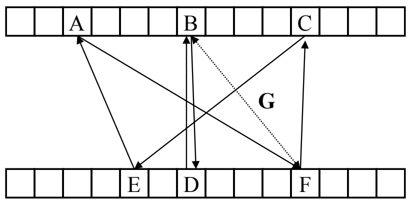
Oblivious Construction of Hash Tables Our schemes use hash tables as read-only look-up tables. Hence, after all items are inserted in a hash table H, only read queries are issued to H. Moreover, all items to be inserted in H are known before H is built. The read phase for item with a key x is simple and consists of reading a bucket that corresponds to $h(x)$ index, where h is a hash function used for the table H. The construction of H is trickier since our schemes require a data-oblivious version of this algorithm.

We use the data-oblivious construction of a hash table H from [21] which uses several calls to oblivious sorting. Its performance hence depends on client member and sorting algorithm used (see Section 3.6).

Cuckoo Hash Tables Pagh and Rodler [42] introduced a cuckoo hashing scheme consisting of two tables, each with m cells, with each cell capable of holding a single key. We make use of two hash functions h^1 and h^2 that we assume can be modeled as completely random functions. The tables store up to n items, where $m = n(1+\psi)$ for some constant $\psi > 0$, yielding a load of (just) less than 1/2; keys can be inserted or deleted over time as long as this restriction is maintained. An item with a key x that is stored in the hash tables must be located at either $h^1(x)$ or $h^2(x)$. As there are only two possible locations for a key, lookups take constant time. We insert a new item (x, v) in the cell $h^1(x)$. If the cell had been empty, the operation is complete. Otherwise, key y previously in the cell is moved to $h^2(y)$. This may in turn require another key to be moved, and so on, until a key is placed in an empty cell. We say



(a)



(b)

Figure 3.1: (a) The top of the figure represents a cuckoo hash table. Keys are placed in one subtable; the arrow for each key points to the alternate location for the key in the other subtable. Key G is inserted, leading to the movement of several other keys for G to be placed, as shown in the bottom of the figure. (b) Key G is to be inserted, but it cannot be placed successfully. (Seven keys have only six locations.) This leads to a failure, or if there is a stash, then G can be placed in a stash.

that a failure occurs if, for an appropriate constant c, after $c \log n$ steps this process has not successfully terminated. Suppose we insert an nth key into the system. It is known that:

- The expected time to insert a new key is bounded above by a constant (that depends on ψ).
- The probability that a new key causes a failure is $\Theta(1/n^2)$ (also depends on ψ).

See Figures 3.1a and 3.1b for examples.

There are several natural variations of cuckoo hashing, many of which are described in a survey article by Mitzenmacher [38]. For our purposes, it suffices to understand standard cuckoo hashing, along with the idea of a *stash* [32].

A stash represents additional memory where keys that would cause a failure can be placed in order to avoid the failure; with a stash, a failure occurs only if the stash itself overflows. As shown in [32], the failure probability when inserting the nth key into a cuckoo hash table can be reduced to $O(1/n^{c'+2})$ for any constant c' by using a stash that can hold c' keys. Using this allows us to use cuckoo hash tables for any polynomially bounded number of inserts and deletions using only a constant-sized stash. To search for an item, we must search both the two table locations and the c' stash locations.

Similar to hash tables, we build cuckoo hash tables by a priori knowing all elements to be inserted into the table. Hence, the life cycle of a cuckoo hash table in our construction is as follows: build a cuckoo hash table with a stash that can fit n elements and then query the table until the next rebuild is due, rebuild the table and the stash using a new pair of hash functions and possibly new elements, access until the next rebuild and so on. The number of times a table can be queried between the rebuilds will be defined in our constructions.

We use the construction of Goodrich and Mitzenmacher [24] to obliviously build a cuckoo hash table with a stash. This construction proceeds by building a bipartite graph, referred to as *cuckoo graph*, where a vertex in the first set corresponds to a cell of the first hash table of the cuckoo table and, similarly, a vertex in the second set corresponds to a cell in the second hash table. Hence, each vertex set is of size m. For every element (x, v) to be inserted into the cuckoo table an edge $(h^1(x), h^2(x))$ is added to the graph. Since in our scheme all elements to be inserted are known, the corresponding cuckoo graph can be constructed. The next phase proceeds by finding connected components of the graph and cycles within each component. Components with cycles that cannot be avoided by removing one of the edges result in moving elements that correspond to cycle edges of this component to the stash. Each step of this algorithm proceeds using several iterations of oblivious sorting within a MapReduce-like framework. For a detailed description of the construction we refer the reader to [24].

Hash Functions We note that for our constructions we will assume that hash functions used in the above data structures are truly random functions, which we can instantiate with calls to pseudorandom functions. The analysis of cuckoo hashing relies on $\log n$-wise independent hash functions [42]. Random functions already posses this property since every new call to a function generates a value that is random and independent of previous calls, unless a call is made for the input that was queried before. A similar argument holds for PRFs.

3.5 Data-Oblivious Memory

In this section we first define the notion of *data-obliviousness* with a running example of a primitive oblivious RAM simulation algorithm. We then show how one measures performance of data-oblivious algorithms, followed by several data-oblivious sorting algorithms.

3.5.1 Definition

Let A be an array consisting of n elements where each element $A[i]$ is of equal size. We say that A that supports the query set $\mathcal{Q}_A$ defined as follows. $\mathcal{Q}_A$ includes $\mathsf{read}(i)$ and $\mathsf{write}(i, \cdot)$ queries to access and modify elements of $A[i]$ for indices $i \in \{0, \dots, n-1\}$.

We require that an element written by write is of the same size as the original element in $A[i]$.

We also define a dataset A and a set of queries $\mathcal{Q}_A$ that is supported by A. A query $q \in \mathcal{Q}_A$ is executed using a call $(A', r) \leftarrow \mathsf{apply}(A, q)$ which returns a possibly modified dataset A' and a reply r to the query q according to A. We overload apply to be able to take a sequence α of operations and apply each operation in a sequence. The final result is then the result of the last query of α and the dataset A' that captures changes of all queries in the sequence. For example, an array A defined above is also a dataset A where apply, depending on the query, either returns a read value or overwrites an existing value in A.

Definition 1 (RAM Simulation Algorithm)**.** *We consider an algorithm $\mathcal{V}$ that has access to a state δ and consists of two functions* setup *and* $\mathsf{simulate}$ *defined below.*

- *(Setup) Let A be the input array where $\mathcal{Q}_A$ is a set of* virtual *queries. The setup function* $\mathsf{setup}(A)$ *returns a digest M and a dataset D (e.g., D could be a rearranged encrypted array A), where $\mathcal{Q}_D$ is a set of* real *queries.*
- *(Simulate) The execution function* $\mathsf{simulate}(D, q)$ *takes a* virtual *query $q \in \mathcal{Q}_A$ and the dataset D, and returns a tuple (D', r, α) where r is an answer to query q, i.e., $(\cdot, r) = \mathsf{apply}(A, q)$, α is a sequence of all* real *queries that were applied to D by $\mathcal{V}$, and D' is a dataset that results from applying queries in α to D.*

To distinguish between virtual queries to A and real queries to D, we refer to queries made to A as *requests*, and queries to D as *accesses*.

Example For the above example D could simply be a copy of A. However, if a dataset owner wishes to hide the content of A she could encrypt A to produce a dataset $D = \{\mathsf{Enc}(A[i]) \mid i \in \{0, \ldots, n-1\}\}$. Then $\mathcal{Q}_D$ is also just a sequence of $\mathsf{read}(i)$ and $\mathsf{write}(i, \cdot)$ on the indices i, while the state M stores the secret key of the underlying encryption scheme.

Example We now describe the above algorithm in the remote storage setting. The user possesses an array A that she wishes to outsource to a remote data storage provider. Instead of storing A with the storage provider, she stores D, e.g., the encrypted array A. When she wishes to access $A[i]$ she sends a query $\mathsf{read}(i)$ to the server and the server returns element $D[i]$ back. The user keeps state M private since M contains the secret key of the encryption scheme she uses.

We adopt the indistinguishability security game from [50] that considers an adversary who can adaptively interact with the simulation algorithm on queries of his choice (IND-CQA Security Game).

Definition 2 (IND-CQA Security Game)**.** *A PPT adversary $\mathcal{A}$ provides the initial data array A. The algorithm $\mathcal{V}$ runs $(D, M) \leftarrow \mathsf{setup}(A)$. She then flips a secret random bit b and returns D to $\mathcal{A}$. $\mathcal{V}$ keeps M private.*

$\mathcal{V}$ and $\mathcal{A}$ then engage in the following process, repeated polynomially many times t:

1. *$\mathcal{A}$ gives $\mathcal{V}$ two queries $q_0, q_1 \in \mathcal{Q}_A$.*
2. *$\mathcal{V}$ executes query q_b, that is $(r, \alpha, D) \leftarrow \mathsf{simulate}(D, q_b)$.*
3. *The algorithm $\mathcal{V}$ returns final dataset D and transcript α to the adversary.*

After t rounds, $\mathcal{A}$ guesses b.

Definition 3 (Data-Oblivious RAM Simulation Algorithm)**.** *We say that algorithm $\mathcal{V}$ is data-oblivious if $\mathcal{A}$'s advantage in winning IND-CQA security game over a random guess is negligible in k, where k is the security parameter.*

We note that an adversary $\mathcal{A}$ of the IND-CQA security game can first "train" himself by observing how $\mathcal{V}$ performs on a single sequence of queries by setting $q_0 = q_1$ in each round. He can then challenge $\mathcal{V}$ on two difference sequences of requests by giving different q_0 and q_1 in at least one round that follows the training phase. After observing the behavior of $\mathcal{V}$ based on the secret bit b she picked, $\mathcal{A}$ can continue interacting with $\mathcal{V}$ by setting $q_0 = q_1$ in each round, before making his guess about b.

The above game captures the security of a simulation algorithm against a curious server in the cloud storage model as follows. In this game, the inputs and outputs of $\mathcal{V}$ that are revealed to the server in the cloud storage model are also revealed to $\mathcal{A}$. However, the secret state M kept by the client, any updates to it and computations inside of $\mathcal{V}$ are kept private, since in the cloud model they are also hidden and happen on the user side.

Example Let us continue with the above example and describe our first oblivious algorithm for accessing a dataset A (see Algorithm 1 for the pseudo-code). Here, in order to hide which index query q is accessing, $\mathcal{V}$ reads and rewrites the whole array regardless of the index of the query q. Hence, α always contains the same set of operations of $\mathcal{Q}_D$. Note that we re-encrypt every value to hide whether the query was read or write. Recall that Enc and Dec are algorithms of a semantically secure encryption schemes, hence, re-encrytion of the same value will result in different

Algorithm 1 An oblivious algorithm for accessing an array A with $O(n)$ access overhead.

```
setup($A$):
    $D \leftarrow [n]$
    for $\forall i \in \{0, \ldots, n-1\}$
        $D[i] \leftarrow \mathsf{Enc}(A[i])$
    end for
simulate($D, q$):
    $\alpha \leftarrow \{\}$
    $e \leftarrow \perp$
    for $\forall j \in \{0, \ldots, n-1\}$
        $(\cdot, z) \leftarrow \mathsf{apply}(D, \mathsf{read}(j))$
        $\alpha \leftarrow \alpha.\mathsf{append}(\mathsf{read}(j))$
        if $q = \mathsf{read}(i)$
            $e \leftarrow \mathsf{Dec}(z)$
        end if
        if $q = \mathsf{write}(i, y)$
            $z' \leftarrow \mathsf{Enc}(y)$
        else
            $z' \leftarrow \mathsf{Enc}(\mathsf{Dec}(z))$
        end if
        $(D, \cdot) \leftarrow \mathsf{apply}(D, \mathsf{write}(j, z'))$
        $\alpha \leftarrow \alpha.\mathsf{append}(\mathsf{write}(j, z'))$
    end for
    return $(e, \alpha, D)$
```

ciphertexts. $\mathcal{V}$ is data-oblivious since regardless what q_b is, the access sequence α is the same.

3.5.2 Performance Measures

We measure the overhead of a data-oblivious algorithm using several parameters:

Access overhead is the number of additional accesses $\mathcal{V}$ is required to do to execute query q obliviously, i.e., the length of the access sequence α. We will differentiate between average-case and worst-case access overhead.

Private memory size m measures space required to be kept secret from $\mathcal{A}$ to perform oblivious accesses, i.e., $m = |M|$. We distinguish between two types of private memory: *state* and *scratch space*. The former is used to keep permanent information that $\mathcal{V}$ requires every time she runs exec. The scratch space is used only while exec is running and is erased as soon as exec returns. We will be interested in sizes of both types of private memory.

Message size is the maximum number of items that can be requested from a dataset D in a single query $q \in \mathcal{Q}_D$, which is also the maximum number of items sent between the user and the server in a single operation.

Space overhead measures additional space required to store D versus storing the original array A.

Example We analyze the algorithm in Algorithm 1 using the above metrics. The algorithm has $2n$ access overhead since it is required to read and write n elements. Note that the user keeps the encryption secret key in the stateful part of M, while element e is stored in the scratch space during the data request. Hence, the size of private memory is constant. The message size is also constant since only one element of D is being read or written at a time. The size of D is the same as the size of A, hence, there is no space overhead.

3.6 Data-Oblivious Sorting

Most algorithms are not data-oblivious since their efficiency depends on making optimizations based on the data-layout. For example, quick-sort is not oblivious since the pattern in which it accesses the data depends on the value and the position of the chosen pivot value and relative order of the data to be sorted.

AKS network [2] and Batcher's [5] sorting networks are examples of data-oblivious sorting. Their access pattern is independent of the data that is being accessed. AKS network sorts n elements in $O(n \log n)$ accesses and Batcher's sorting network in $O(n(\log n)^2)$. However, the constant is high in the former algorithm [9], making the Batcher's sorting network more efficient in practice.

In Batcher's sorting network the data is accessed two elements at a time, swapped if necessary, and written back. Goodrich and Mitzenmacher [24] showed that the running time of this algorithm can be improved significantly if more than two elements are read at a time. In particular, if m is the size of the private memory M where elements are read, sorting could be performed in $O(\frac{n}{m}(\log_m n)^2)$ time. Hence, if the size of private memory is $m = O(\sqrt{n})$, then sorting can be done in $O(n)$ time.

Goodrich [23] proposed a randomized data-oblivious Shellsort that runs in time $O(n \log n)$ using client memory of size $O(1)$. It differs from the above sorting network methods since it sometimes fails to produce the correct results, i.e., a sorted sequence. However, the probability of this happening is inverse of a polynomial in n.

3.7 Square-Root Oblivious Access Scheme [18, 21]

We present an overview of the square-root oblivious RAM simulation method [18] as a warm-up for our oblivious RAM schemes. We give enough details about the method for a reader to understand our de-amortized version provided in Section 6.1 and refer the reader to [18] for the full description.

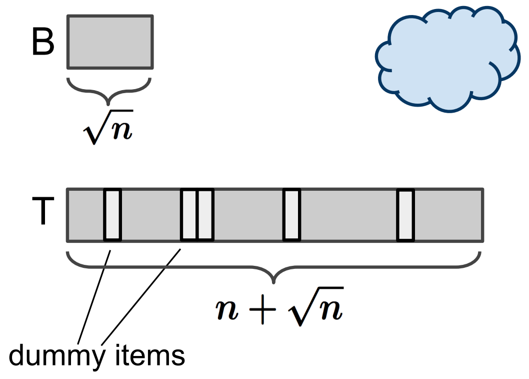


Figure 3.2: Memory layout of the storage provider during oblivious RAM simulation, original version of the square-root solution [21].

Data Layout Let A be the original dataset containing n items. The square-root solution stores a dataset D of size $n+2\sqrt{n}$ at the storage provider (aka server). D is split into a buffer, B, and a table, T. The buffer B has size $\sqrt{n}$ and is used to cache the last $\sqrt{n}$ requests. Table T contains a random permutation of the n encrypted data items of A and $\sqrt{n}$ dummy items. Every time the user sends a read request to the server we assume that she decrypts the result, and re-encrypts an item when writing it back. (See Figure 3.2.)

Each data item of A is associated with a key (virtual address of) x, $x = 1, \cdots, n$ and each dummy item is given a key $n + d$ where $d = 1, \cdots, \sqrt{n}$. All items in T are ordered according to a pseudorandom permutation function π such that $\pi(x)$ gives the location of the item with key x in T. (The full solution, which uses a binary search, is omitted in our description since the overall complexity of the request is $O(\sqrt{n})$.)

The user keeps the encryption key, the seed of the current permutation π, and the counter of the number of requests made to the store, request_count, in her private memory M. Hence, the size of the private memory is constant.

Request Simulation The square-root ORAM simulation method is outlined in Algorithm 2. The simulation of a data requests consists of two phases *access* and *rebuild* phases. The access phase consist of scanning buffer B and a read access to table T, followed by the write access to B. When the user writes to the last free cell of B, the user enters the rebuild phase which involves emptying B into T and rebuilding T. Note that table T has to be rebuilt after every $\sqrt{n}$ requests. The rebuild phase consists of obliviously replacing the items in T for which there is a new instance in B, associating the keys of real and dummy items with tags from a new permutation π', and sorting items in T according to π'. The rebuild phase takes $O(n \log^2 n)$ accesses. Since the rebuild happens only once every $\sqrt{n}$ requests, the amortized access overhead per request is $O(\sqrt{n} \log^2 n)$: $O(\sqrt{n})$ accesses for the request phase and $O(\sqrt{n} \log^2 n)$ accesses for the rebuild phase.

Analysis Requests are handled by scanning buffer B and accessing T. Due to the scheduled rebuilds, data items are associated with new tags every $\sqrt{n}$ requests. Between the rebuilds, unique locations are accessed in T: either an item x is not present in the buffer and hence a unique location, $\pi(x)$, is accessed, or a unique dummy item, $\pi(n + \text{request\_count})$, is accessed.

Algorithm 2 Oblivious RAM simulation using the square-root approach [21].

```
Generate pseudorandom permutation function $\pi$
Initialize table $T$ by storing the $n$ data items and $\sqrt{n}$ dummy items according to permutation $\pi$
request_count $\leftarrow 1$
**while** true **do** {process a request}
    found $\leftarrow false$
    Scan all the locations in buffer $B$. During the scan, if data item $x$ is found, set found $\leftarrow$ true.
    **if** found **then**
        Access location $\pi(n + \text{request\_count})$ in $T$ {dummy item}
    **else**
        Access location $\pi(x)$ in $T$ {data item $x$}
    **end if**
    Rewrite $B$, adding or replacing data item $x$
    request_count $\leftarrow$ request_count $+ 1$
    **if** request_count $> \sqrt{n}$ **then**
        Generate pseudo-random permutation function $\pi'$
        Construct a new table $T'$ with $\pi'$ using items in $T$ and $B$.
        clear $B$ and set $T \leftarrow T'$, $\pi \leftarrow \pi'$, and request_count $\leftarrow 1$
    **end if**
**end while**
```

Chapter Four

The $\log n$-Hierarchical Oblivious Access Scheme

In this section we present a solution that improves over the square-root solution (Section 3.7) and achieves $O(\log n)$ amortized access overhead assuming the user has private memory of size $O(n^\epsilon)$, for some constant $\epsilon > 0$. Similarly to the square-root solution we use a buffer, called a cache, to record previous requests and avoid accessing the same memory cells in case these items are requested again. However, instead of one cache our solution makes use of a hierarchy of caches, each implemented as either a hash table with buckets or a cuckoo hash table with a common stash.

4.1 Setup and Data Layout at the Server

Following Definition 1, we describe the `setup` function and its output of dataset D, following the definition of an oblivious simulation scheme. Let n be the number of memory cells of the original dataset A. We view each such cell as an item consisting of a pair (x, v), where $x \in \{0, \cdots, n-1\}$ is the index and v is the corresponding value. Our data structure D stored at the server has four components, illustrated in Figure 4.1. The first component is a cache of size $\alpha \log n$, denoted by C. The second component is a hierarchy of hash tables with buckets of size $c \log n / \log\log n$, $H_1, \ldots, H_l$ where $c \geq 1$ is a constant and l is $O(\log\log n)$ (the choice for l will become evident from the analysis). H_1 can fit twice more elements than C, H_2 fits twice more elements than H_1 and so on. The third component is a hierarchy of cuckoo hash tables, $T = T_{l+1}, \ldots, T_L$, where T_{l+1} is suitable to fit twice more elements than H_l. Each consequent table T_{j+1} is twice the size of table T_j, and T_L is the first table in the sequence of size greater than or equal to n. Thus, L is $O(\log n)$. The fourth component is a stash, S, which is shared between all the above cuckoo tables. We refer to a hash table H_i and a cuckoo hash table T_j using indices $i \in \{1, \ldots, l\}$ and $j \in \{l+1, \ldots, L\}$.

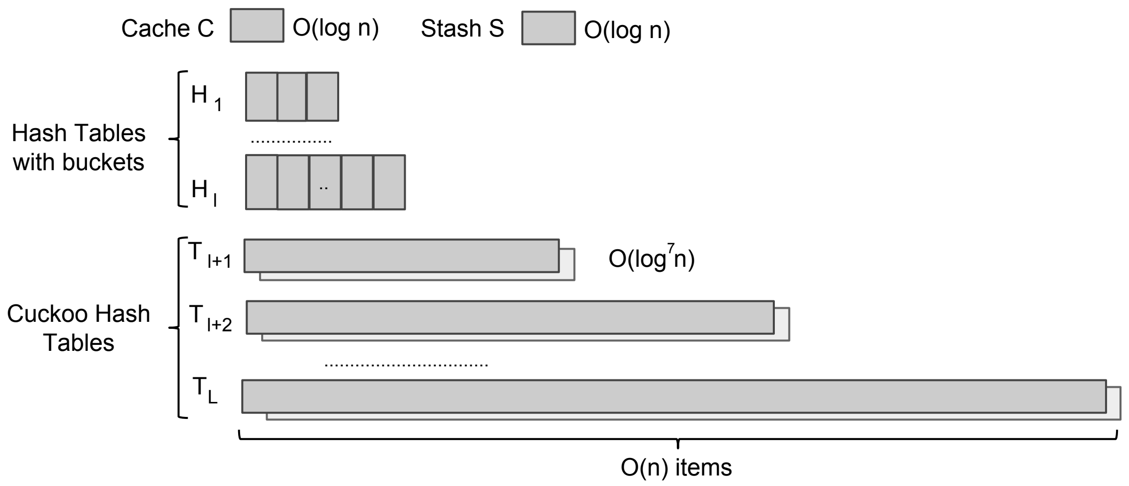


Figure 4.1: Illustration of the data structure stored at the server for oblivious RAM simulation with $O(\log n)$ amortized access overhead. In the access phase of the simulation, all the items in the cache, C and the stash, S, a bucket in hash tables H_i, plus two items for each cuckoo table T_j are read by the server.

The items in D are stored in the data structure in an encrypted form. We use a semantically secure encryption scheme. Recall that with this scheme re-rencyption of the same item is very likely to produce a different ciphertext (see Section 3.3). Hence, the server is unable to determine whether two ciphertexts correspond to the same item or not.

The items are placed in a hash table H_i according to a function f_i, for $i \in \{1, \ldots, l\}$. In particulate an item (x, v) is mapped to a bucket $f_i(x)$ in H_i. The items are inserted in a cuckoo hash table T_j using two functions, h_j^1 and h_j^2, for $j \in \{l+1, \ldots, L\}$. Recall that in a cuckoo hashing scheme with a stash an item (x, v) is located in one of the two locations in T_j, $h_j^1(x)$ or $h_j^2(x)$. We instantiate hash function f_i (and similarly h_j^1 and h_j^2) with a family of pseudorandom functions parameterized by a secret value, k_i, for each hash table, H_i, such that value k_i is not revealed to the server. In particular, k_i is stored in an encrypted form for each table H_i, so that each user can read k_i, decrypt it, and then use it to instantiate a hash function, f_i.

The content of the tables in the construction consists of real, fake and dummy elements. The real elements are the original elements from the dataset A with keys in the set $\{0, \ldots, n-1\}$. The fake elements are added to every level of the construction and are used during the access phase to avoid querying real elements more than once from this level. We add $2^i \times \alpha \log n$ fake elements to level i in the same manner for hash tables and cuckoo hash tables. A key of the fake element is in the range $\{n, \ldots, 2^i \times \alpha \log n - 1\}$. Overall, each level i is large enough to fit $2^{i+1} \times \alpha \log n$ elements, fake and real. During the simulation we need to hide the number of real elements at every level. For this purpose, each table is padded with dummy elements such that table size is static and independent of how many real elements there are in the table. For example, hash table H_i has $2^{i+1} \times \alpha \log n \frac{c \log n}{\log \log n}$ cells, due to $c \log n / \log \log n$-sized buckets, but contains at most of $2^{i+1} \times \alpha \log n$ elements. We fill empty cells with dummy elements and encrypt them as well. Similarly, a cuckoo hash table at level j contains two tables of size $O(2^{j+1} \times \alpha \log n)$ each but is required to store at most of $2^{j+1} \times \alpha \log n$ real and fake elements. Once again, we use probabilistic encryption scheme which makes it hard for the adversary to distinguish whether a cell contains a real, a fake or a dummy element.

During the runtime we also keep a counter request_count of the number of requests that have been done to the simulator. We will show that in order to simulate requests obliviously, clients need to know request_count. However, since this counter is small it can be stored at the server encrypted and accessed every time a request is made, leaving the system stateless.

The data structure is initialized by storing all the n RAM items into cuckoo table T_L. However, the scheme can be adapted to the case when no items are stored in T_L or when items with sequential indices are added to the store.

4.2 Oblivious Simulation of Data Requests

We split the sequence of accesses the algorithm produces in order to simulate a data request obliviously into two phases, an *access phase* and a *rebuild phase*.

4.2.1 Access Phase

Suppose the user calls for an access to memory item with a key x. The access phase consists of a search for x in the cache, C, then in the stash, S, and continues with sequential lookups in hash tables $H_1, \ldots, H_l$, followed by two lookups in every cuckoo hash $T_{l+1}, \ldots, T_L$. The lookup in each table depends on whether the item has been found in C, S or earlier tables, or not. If the item was not found in C or S, the bucket at location $f_1(x)$ is accessed in T_1. The bucket is scanned in search for x. We follow the same strategy for the consequent hash tables until the item is found. If the item was not found in hash tables $H_1, \ldots, H_l$, cuckoo hash table T_{l+1} is accessed in locations $h^1_{l+1}(x)$ and $h^2_{l+1}(x)$, and similarly for consequent T_j's, if x is not found in earlier tables.

Once the item is found, in either H_i or T_j we continue accessing consequent tables but at random locations instead of following functions f and h. Note that although the item is found we have to continue accessing every table, since revealing at which level the item is found may reveal what item was being searched for. In Section 4.3 we show that accesses that result from looking for an item are indistinguishable from the ones that are random.

Once we have completed the access phase, which takes $O(\log n)$ time, we then switch to the rebuild phase. We begin by adding or replacing a copy of the found item into cache C, possibly changing its value in the case of a write operation. To assure obliviousness, we exhaustively scan C in a sequential manner and re-encrypt and rewrite all its items. Thus, the server cannot distinguish which item was accessed and whether it was modified.

We note briefly that if the item is in the stash, we can obliviously remove it from the stash when placing it into C, to help make sure the stash does not overflow. One natural approach is to have stash cells have an associated "clean" or "dirty" bit, which is encrypted along with the rest of the item. A clean cell can store an item; a dirty cell is currently being utilized. When an item is found and replaced into C, we can set the cell to clean in the stash.

4.2.2 Rebuild Phase

After adding enough items, cache C will eventually overflow. Specifically, let $|C| = \alpha \log n$. We remedy the overflow by moving all the elements of C to hash table H_1 after every $\alpha \log n$ requests, i.e., when $\text{request\_count} \mod c \log n = 0$. The moving down of elements cascades down through the hierarchy of hash tables at a fixed schedule by periodically moving the elements of level $i-1$ into H_i at the earliest time H_{i-1} could have become full. Now suppose that we are going to move elements into table H_i for the second time, then we instead move the elements into table H_{i+1}. Moreover, we continue applying this rule for $i = 1, 2, \ldots l$, until we are copying the elements into a table for the first time or we reach H_l. Once H_l is reached we move elements to T_{l+1}, and continue similar table cascade but for cuckoo hash tables. The cascade of rebuilds stops, once T_L is reached. Note that rebuilds of T_j tables need to take into account elements from stash S, i.e., elements that did not fit in T_{j-1} the last time it was built. Thus, the process of copying elements into a (cuckoo) hash table occurs at deterministic instances, depending only on the number of requests r that have been specified by algorithm $\mathcal{A}$ so far.

In order to move m elements from level i, $0 \leq i < l$ into a hash table H_{i+1} obliviously, we use an algorithm of [24] to obliviously sort the items using $O(m)$ accesses to the outsourced memory, assuming we have a private workspace of size $O(n^\epsilon)$, for some constant $0 < \epsilon < 1$, and $m \geq \log n$, which is always true in our case. Constructing a cuckoo hash table T_j, $l+1 \leq j \leq L$, is trickier and we use another algorithm of [24] to obliviously construct a cuckoo table of size m and an associated stash, S', of size $O(\log n)$ in $O(m)$ time, with very high probability, while utilizing the private workspace of size $O(n^\epsilon)$. Given this construction, we then read S and S' into our private workspace, remove any duplicates and merge them into a single stash S (which will succeed with very high probability, based on Theorem 5), and write S back out in a straightforward oblivious fashion. Note that in order to assure obliviousness in subsequent lookups, hash table H_{i+1} (cuckoo hash table T_{j+1}) is rebuilt using a new pseudorandom function (two new pseudorandom functions) selected by the client by replacing parameter k_{i+1} (k_{j+1}) with a new one.

4.3 Analysis

We first prove that hash tables and cuckoo hash tables used in the solution do not overflow with very high probability and, hence, can be safely used for oblivious simulation. We then show that access sequence appears to be independent of the data to the server and, hence, our construction is data-oblivious following Definition 3.

Lemma 4. *Let H be a hash table with m buckets of size $\frac{c \log n}{\log \log n}$ for an arbitrary $c \geq \mathrm{e}$ and m between $\Omega(\log n)$ and $O((\log n)^7)$. Then no bucket of H overflows after m*

elements are inserted in an empty H with a very high probability is bounded by $1 - 1/n^{c'}$ where c' is an arbitrary constant.

Proof. We use balls and bins analysis to bound the overflow event, since inserting m elements in m buckets is equivalent to throwing m balls into m bins and computing the maximum number of balls in any bin.

We want to prove that buckets of size $s = \frac{c \log n}{\log \log n}$ are enough to avoid overflow with very high probability. We use analysis similar to [39, Chapter 5], where the probability of having at least s balls in any bin is bounded by

$$p_s = m \left(\frac{\mathrm{e}}{s}\right)^s$$

If $s = \frac{c \log n}{\log \log n}$ then

$$p_s \leq m \left(\frac{\mathrm{e} \log \log n}{c \log n}\right)^{\frac{c \log n}{\log \log n}}$$

Since m is at most $O((\log n)^7)$ in our case we bound the expression further:

$$\begin{aligned} p_s &\leq \log^7 n \left(\frac{\mathrm{e} \log \log n}{c \log n}\right)^{\frac{c \log n}{\log \log n}} \\ &\leq 2^{\log \log^7 n + \frac{c \log n}{\log \log n} \log\left(\frac{\mathrm{e} \log \log n}{c \log n}\right)} \end{aligned}$$

Let $c_1 = c/\mathrm{e}$ then:

$$\begin{aligned} p_s &\leq 2^{\log \log^7 n + \frac{c \log n}{\log \log n} \log\left(\frac{\log \log n}{c_1 \log n}\right)} \\ &\leq 2^{\log \log^7 n + \frac{c \log n \log \log \log n}{\log \log n} - \frac{c \log n \log c_1 \log n}{\log \log n}} \\ &\leq 2^{\log \log^7 n + \frac{c \log n \log \log \log n}{\log \log n} - c \log n} \\ &\leq 2^{\frac{c \log \log \log n}{\log \log n} \log n - (c-1) \log n} \end{aligned}$$

If $\log\log n / \log\log\log n \geq c$ then $\frac{c\log\log\log n}{\log\log n}\log n \leq \log n$. Otherwise, there is such a $c_2 < c$ that $\frac{c\log\log\log n}{\log\log n}\log n \leq c_2 \log n$. Hence, there is a constant $c' = c - c_2$ that

$$m\left(\frac{\mathrm{e}}{s}\right)^s \leq \frac{1}{n^{c'}}$$

$\square$

Lemma 5. *Let $T_{l+1}, \ldots, T_L$ be a sequence of cuckoo hash tables where T_l contains $O((\log n)^7)$ elements, $|T_{j+1}| = 2|T_j|$ and $|L| = O(\log n)$. A stash of size $O(\log n)$ shared between $T_{l+1}, \ldots, T_L$ is enough to avoid overflows with very high probability.*

Proof. The probability that the stash for a cuckoo hash table of size x cells (where x is $\Omega(\log^7 n)$) exceeds a total size s is $x^{-\Omega(s)}$ [24]. Further, as long as the hashes for a cuckoo hash table at each level are independent, we can treat the required stash size at each level as independent, since the number of items placed in the stash at a level is then a random variable dependent only on the number of items appearing in that level.

Now consider any point of our construction and let S_i be the number of items at the ith level that need to be put in the stash. It is apparent that S_i has mean less than 1 and tails that can be dominated by a geometrically decreasing random variable. This is sufficient to apply standard Chernoff bounds. Formally, let $X_1, X_2, \ldots, X_L$ be independent random variables with mean 1 geometrically decreasing tails, so that $X_i = j$ with probability $1/2^j$ for $j \geq 1$. Then the calculations of [24] imply that the X_i stochastically dominate the S_i, and we can now apply standard Chernoff bounds for these random variables. Specifically, noting that X_i can be interpreted as the number of fair coin flips until the first heads, we can think of the sum of the X_i as being the number of coin flips until the ℓth head, and this dominates the number of items that need to be placed in the stash at any point. Since $L = O(\log n)$ then for any constant γ_1 there exists a corresponding constant γ_2 such that the Lth head occurs by the $(\gamma_2 \log n)$'th flip with probability at least $1 - 1/n^{\gamma_1}$. (See, for example, [39, Chapter 4].) Hence,

$$\Pr\left(\sum_{i=1}^{L} S_i > L\right) \leq 1 - \Pr\left(\sum_{i=1}^{L} X_i \leq \gamma_2 \log n\right) \leq 1/n^{\gamma_1}.$$

Therefore we can handle any polynomial number of insertions with high probability, using a stash of size only $O(\log n)$ that holds items from all levels of our construction.

$\square$

Theorem 6. *The* $\log n$*-hierarchical oblivious RAM simulation of memory of size* n *presented in this chapter has an amortized access overhead of* $O(\log n)$ *using* $O(1)$ *space overhead and assuming that a client has access to private workspace of size* $O(n^{\epsilon})$*, for* $\epsilon > 0$*, with very high probability.*

Proof. We analyze the overhead of the access phase and the rebuild phase. Recall that during the access phase every part of the data structure is accessed either for a real or a dummy element, regardless of where the element of interest has been found. Hence, we compute the total number of accesses one has to do to perform oblivious simulation, that is, the sequence of accesses that starts with scanning the cache and ends with two accesses to the last cuckoo hash table H_L, rewriting the cache and the stash.

The access phase of the simulation begins with scanning the entire cache and the stash, both of size $O(\log n)$. It is then followed by accessing a bucket of size $c \log n / \log \log n$ from every hash table $H_1, \ldots, H_l$, where $l = O(\log \log n)$. Hence, accessing hash tables takes $O(c \log n / \log \log n \times \log \log n) = O(\log n)$ calls to server.

Accessing cuckoo hash tables is also $O(\log n)$ in total; two accesses in every table $T_{l+1}, \ldots, T_L$ where $L = O(\log n)$. , i.e., making $c \log n$ accesses in total, and finally accessing two items in $O(\log n)$ cuckoo hash tables. Finally, the cache and stash are rewriting, also giving $O(\log n)$ accesses. Summing up accesses to all the parts of the construction we get $O(\log n)$ overhead per every access.

We now analyze the rebuild phase. Each hash table H_i is rebuilt after the previous table is full, starting with the rebuild of H_1 when the cache is full. Hence, H_i is rebuilt after $2^i \times \alpha \log n$ simulated requests, where $\alpha \log n$ is the size of the cache. The rebuild of H_i consists of constant number of sorting passes using the method of [24] with n^ϵ private memory, for $\epsilon < 1$, where each pass takes time linear in the size of the table including buckets, $O(2^{i+1} \log n \times \log n / \log \log n)$ accesses. Since the rebuilds are scheduled and always happen after $2^i \times \alpha \log n$ requests, the cost of the rebuild of every table can be amortized over the requests that cause each level to be full:

$$\sum_{1 \le i \le l} O\left(2^{i+1} \log n \times \frac{\log n}{\log \log n} \times \frac{1}{2^i \log n}\right) = O(\log n)$$

Similarly, for cuckoo hash tables we use the method of [24] to construct every table T_j in $O(2^j \log n)$ accesses. Every rebuild of T_j is scheduled after $2^{j+1} \times \alpha \log n$ simulated requests. The total amortized cost for rebuilding $O(\log n)$ cuckoo hash tables is then:

$$\sum_{l+1 \le j \le L} O\left(2^{j+1} \log n \times \frac{1}{2^j \log n}\right) = O(\log n)$$

Hence, the rebuild phase has $O(\log n)$ amortized overhead, giving $O(\log n)$ amortized access overhead to the scheme.

The rebuild of a hash table can fail if a bucket of a hash table receives more than $c \log n / \log \log n$ elements. However, in Lemma 4 we show that this event happens with $1/n^{c'}$ probability for some constant $c' \geq 1$. The rebuild of a cuckoo hash table can fail if the number of elements that are required to put in the stash exceeds $O(\log n)$. The result of Lemma 5 shows that this event also happens with probability $1/n^{\gamma_1}$ for some constant γ_1. Hence, the $\log n$-hierarchical oblivious simulation has $O(\log n)$ overhead with very high probability. □

Theorem 7. *The* $\log n$*-hierarchical RAM simulation presented in this chapter is a data-oblivious access simulation scheme as per Definition 3.*

Proof. We analyze the sequence of accesses $\mathcal{L}$ made by the algorithm during the simulation of r requests (request_count in the pseudo-code) on an n-size dataset A and show that it can be split into two types of accesses: deterministic and probabilistic accesses. We first show that the subsequences that correspond to the deterministic accesses depend only on r and n. We then show that probabilistic accesses correspond to a uniform distribution and, hence, are independent of the original request sequence. We conclude that independence of $\mathcal{L}$ and the simulated requests relies on the security of PRFs and semantically secure encryption scheme. Hence, a computationally bounded adversary $\mathcal{A}$ has a negligible advantage in distinguishing which challenge request sequence the algorithm is simulating in the IND-CQA Security Game in Definition 2.

We split the access sequence $\mathcal{L}$ into subsequences that correspond to the access and the rebuild phase. This split can be made easily since the access phase always makes the same number of accesses regardless of the simulated sequence and the rebuild phase accesses are scheduled deterministically based on r and n. We further split each subsequence into accesses that are deterministic in nature. These access subsequences include reading and writing to the cache and stash in the beginning and the end of the access phase for one simulated request. They also include the accesses from the rebuild phase since the oblivious sorting and cuckoo hash table construction algorithms make deterministic accesses to the memory. We note that the adversary knows whether it is a deterministic access or not, and whether an access corresponds to the access or the rebuild phase. However, these can be determined from r and n alone. The deterministic subsequences are, hence, data-oblivious by nature since they do not depend on the real data being requested.

We now consider the rest of the sequence $\mathcal{L}$, i.e., accesses that are probabilistic. These accesses result from accessing hash tables and cuckoo hash tables during the access phase. We consider the subsequence of accesses after the stash is being scanned and until the cache and stash is rewritten. This sequence consists of reading l buckets of size $c \log n / \log \log n$ from hash tables $H_1, \ldots, H_l$ and two accesses at every cuckoo hash table $T_{l+1}, \ldots, T_L$. We note that the adversary knows which table is being accessed and whether accesses correspond to reading a bucket or reading two locations in a cuckoo hash table, since this split depends on the size of the dataset n and is, hence, deterministic and is independent of the simulated request sequence. We are interested in showing that when the bucket is read from hash tables H_i, it appears to be picked uniformly at random among all $2^i \times \alpha \log n$ buckets of H_i. Similarly for T_j we want to show that an access to the first and second table appears to be uniform over all locations in each table.

Let an epoch of the level i be a a sequence of accesses made to the table at this level between two of its sequential rebuilds. We consider the accesses that happen between the rebuilds since hash functions based on different secret are used to access H_i (T_j) before and after the current epoch. Note that only distinct keys are accessed from H_i, and similarly, T_j, during an epoch. The distinctiveness of the accessed keys is guaranteed from the simulation algorithm. If a real key x is accessed from table i and is not found there, the construction guarantees that it will be found in level $i' > i$. If $i = L$, the item is in T_j by construction. Once found, an item of key x is moved to the cache and, hence, will not be accessed from level i until level $i-1$ has to be rebuilt, or if $i = 1$ the cache becomes full. However, in this case the epoch in which x was accessed from level i is over and level i is rebuilt. Similarly, a fake key x is accessed only once since this key is based on request_count which is incremented for every request. The number of fake keys at level i is the same as the number of real keys, hence, we can accommodate the situation when all items are found in the above levels and no real element is accessed during an epoch of level i.

The hash table H_i has $2^i \times \alpha \log n$ buckets and is accessed $2^i \times \alpha \log n$ times during its epoch. During the epoch, H_i is accessed with buckets that correspond to a distinct set of item keys. This set consists of real keys from the set of real keys $\{0, \ldots, n-1\}$ and set of fake keys $\{n, \ldots, n + 2^i \times \alpha \log n - 1\}$. Note that a bucket could be accessed more than once due to the collisions caused by the hash function on different keys. However, since we use a pseudorandom function (PRF) as a hash function, the bucket assignment appears to be random and ,hence, independent of the key due to the security properties of PRFs.

The analysis of the accesses to cuckoo hash table T_j during its epoch is similar. The accesses to T_j are only made for a distinct set of keys, where each key is either a real key or a fake key. The same location could be accessed more than once due to collisions of the hash function. However, as for hash tables with buckets, a PRF is used to assign keys to locations in the cuckoo hash table during an epoch. Hence, the accessed locations appear to be uniform over all locations in each of the two tables of T_j.

The accesses for fake and real elements in tables H_i and T_j cannot be distinguished by the adversary since these elements are stored encrypted using a semantically secure encryption scheme. Hence, the level i where the item is found and at which the accesses are changed to fake accesses is hidden from the adversary.

Overall, the security of the scheme relies on the security properties of PRFs and an underlying encryption schemes. The adversary has only a negligible advantage in distinguishing between a PRF and a truly random function and, hence, determining the key assignment in the hash tables and cuckoo hash tables. The adversary also has only a negligible advantage in distinguishing whether a ciphertext correspond to a fake or a real element and subsequently determining where the item is found in the construction. Hence, for a computationally bounded adversary the sequence $\mathcal{L}$ appears to be independent of the simulated requests.

$\square$

Algorithm 3 The access and rebuild phase during oblivious access simulation with the $\log n$-hierarchical approach (Chapter 4).

```
{Access phase}
$found \leftarrow false$
scan cache $C$ and stash $S$. if $x$ is found in one of them set $found \leftarrow true$
**for** each level $i$, $1 \le i \le l$ **do**
    if $found$ is $true$, $y \leftarrow n + r \mod 2^i|C|$, else $y \leftarrow x$
    Access bucket $f_i(y)$ in $H_i$.
    if $x$ is found in the bucket, $found \leftarrow true$
**end for**
**for** each level $j$, $l + 1 \le j \le L$ **do**
    if $found$ is $true$, $y \leftarrow n + r \mod 2^j|C|$, else $y \leftarrow x$
    Access locations $h_j^1(y)$ and $h_j^2(y)$ in $T_j$.
    if $x$ is found set $found \leftarrow true$
**end for**
{Rebuild phase}
Rewrite $S$. If $x$ was found in $S$, overwrite $x$ in $S$ with dummy.
Rewrite $C$, adding or replacing data item $x$.
read request counter request_count
$r \leftarrow r + 1$
**if** $(r \mod |C|) = 0$ **then**
  $H_1 \leftarrow$ rebuild$(C, H_1)$
  $C \leftarrow$ empty_cache()
  $k \leftarrow 1$
  {Cascade rebuilds along hash tables}
  **while** $(r \mod 2^k \log n) = 0$ **and** $k \le l$ **do**
    **if** $k = l$ **then**
      $T_1 \leftarrow$ rebuild$(H_k, T_1)$
    **else**
      $H_{k+1} \leftarrow$ rebuild$(H_k, H_{k+1})$
    **end if**
    $H_k \leftarrow$ init_hashtable()
    $k \leftarrow k + 1$
  **end while**
  {Cascade rebuilds along cuckoo hash tables}
  **while** $(r \mod 2^k \log n) = 0$ **and** $k \le l + L$ **do**
    **if** $k = l + L$ **then**
      $T_L \leftarrow$ rebuild$(T_L, S)$
    **else**
      $T_{k+1} \leftarrow$ rebuild$(T_k, T_{k+1}, S)$
      $T_k \leftarrow$ init_cuckoo_hashtable()
    **end if**
    $k \leftarrow k + 1$
  **end while**
**end if**
**return** $x$
```

Chapter Five

Constant Overhead Oblivious Access Scheme

In this chapter, we study *oblivious storage* (OS), a natural way to model privacy-preserving data outsourcing. We show that Alice can hide both the content of her data and the pattern in which she accesses her data, with high probability, using a method that achieves $O(1)$ amortized rounds of communication between her and Bob for each data request. In contrast with Chapter 4, we assume that Alice and Bob exchange small messages of size $O(n^{1/c} \log n)$ in a single round, for $c \geq 2$. We also assume that Alice has a private memory of size $n^{1/c} \log n$. These assumptions model real-world cloud storage scenarios, where trade-offs occur between latency, bandwidth, and the size of the client's private memory.

5.1 The Oblivious Storage (OS) Model

We now describe the differences between the RAM storage model we consider in the square-root solution in Section 3.7 and the $\log n$-hierarchical solution in Section 4.

5.1.1 Client Private Memory

We assume that the client has access to a small private memory, M, which is comprised of permanent storage and scratch space. The permanent storage includes the encryption key and the current seed of the permutation the client is using, which together is of size $O(1)$. The rest of M is used as a scratch space while performing operations on the remote storage and is not needed in between operations. We require the size of the scratch space to be sublinear in n, in particular, we will use a private storage of size $n^{1/c} \log n$, for an arbitrary integer $c \geq 2$.

Since the client can store a small number of elements at a time, we assume that he does not try to request from the server more than he can fit and process in M. Let the *message size*, denoted m, be the maximum elements that can be exchanged by the client and server in one operation. We have that m should be less than the size of the scratch space.

5.1.2 Server Storage

The server supports the following operations on an array of data items S. Here, S refers to the name of the file or blob (e.g., in Microsoft Azure) where items are stored at the server. The number of items in a single call is limited by m, the bandwidth between the client and the server and the size of client's private memory, which in our case is $O(n^{1/c} \log n)$

- $\mathsf{get}(S, \mathsf{loc})$: return element stored at a location loc in S.
- $\mathsf{put}(S, \mathsf{loc}, e)$: put element e to a location loc in S.
- $\mathsf{getRange}(S, \mathsf{loc}, \ell)$: return an array a with elements at locations $\mathsf{loc}, \ldots, \mathsf{loc} + \ell - 1$ in S, where $\ell \leq m$.
- $\mathsf{putRange}(S, \mathsf{loc}, a)$: write elements in array a to locations $\mathsf{loc}, \ldots, \mathsf{loc} + |a| - 1$ in S, where $|a| \leq m$.
- $\mathsf{getRangeDist}(S, \langle \mathsf{loc}_1, \ldots, \mathsf{loc}_c, \rangle, \langle \ell_1, \ldots, \ell_c, \rangle)$: return an array a with elements at locations $\mathsf{loc}_i, \ldots,$ $\mathsf{loc}_i + \ell_i - 1$ in S,$\forall i \in [1, c]$, where $\sum \ell_i \leq m$.
- $\mathsf{putRangeDist}(S, \langle \mathsf{loc}_1, \ldots, \mathsf{loc}_c \rangle, \langle a_1, \ldots, a_c \rangle)$: write elements in each array a_i to locations $\mathsf{loc}_i, \ldots,$ $\mathsf{loc}_i + |a_i| - 1$ in S, where $\sum |a_i| \leq m$.

We assume that the server can perform operations get and put in constant time and operations getRange, putRange, getRangeDist and putRangeDist in time proportional to the number of elements read or written, but each operation takes one I/O.

Definition 8 (Metadata)**.** *The name of the array S, its size, location i, l, and the size of a are referred to as the* metadata *of a* getRange *or a* putRange *call. Similarly for* getRangeDist *and* putRangeDist*, locations $\langle \mathsf{loc}_1, \ldots, \mathsf{loc}_c \rangle$, $\langle \ell_1, \ldots, \ell_c \rangle$ and sizes of $a_1, \ldots, a_c$ are referred as metadata as well.*

5.2 Shuffle Method

One of the key techniques in our solutions is the use of oblivious shuffling. The input to any *shuffle* operation is a set, A, of n items. Because of the inclusion of the getRange operation in the server's API, we can view the items in A as being ordered by their keys. Moreover, this functionality also allows us to access a contiguous run of m such items, starting from a given key. The output of a shuffle is a permutation of the items in A with replacement keys, so that all permutations are equally likely. During a shuffle, the server, Bob, can observe Alice read (and remove) m of the items he is storing for her, and then write back m more items, which provides some degree of obfuscation of how the items in these read and write groups are correlated.

During such a shuffle, we assume that Alice is wrapping each of her key-value pairs, (x, v), as $(x', (x, v))$, where x' is the new key that is chosen to obfuscate x. Indeed, it is likely that in each round of communication that Alice makes she will take a wrapped (input) pair, (x', X), and map it to a new (output) pair, (x'', X'), where the X' is assumed to be a re-encryption of X. The challenge is to define an encoding strategy that for the x' and x'' wrapper keys so that it is difficult for the adversary to correlate inputs and outputs.

5.2.1 Oblivious Sorting

One way to do the oblivious shuffle is to assign each item a random key from a very large universe, which is separate and distinct from the key that is a part of this key-value pair, and obliviously sort the items by these keys (see Section 3.6). That is, we can wrap each key-value pair, (x, v), as $(x', (x, v))$, where x' is the new random key, and then wrap these wrapped pairs in a way that allows us to implement an oblivious sorting algorithm in the OS model based on comparisons involving the x' keys. Specifically, during this sorting process, we would further wrap each wrapped item, $(x', (x, v))$, as $(y, (x', (x, v)))$, where y is an address or index used in the oblivious sorting algorithm. So as to distinguish such keys even further, Alice can also add a prefix to each such y, such as "`Addr:`" or "`Addr`i:", where i is a counter (which could, for instance, be counting the steps in Alice's sorting algorithm). Using such addresses as "keys" allows Alice to consider Bob's storage as if it were an array or the memory of a RAM. She can then use this scheme to simulate an oblivious sorting algorithm.

If the randomly assigned keys are distinct, which will occur with very high probability, then this achieves the desired goal. And even if the new keys are not distinct, we can repeat this operation until we get a set of distinct new keys without revealing any data-dependent information to the server.

Shuffling by sorting items via randomly-assigned keys generates a random permutation such that all permutations are equally likely (e.g., see [33]). In addition, since the means to go from the input to the output is data-oblivious with respect to the I/Os (simulated using the address keys), the server who is watching the inputs and outputs cannot correlate any set of values. That is, independent of the set of I/Os, any input set A can be mapped to any output permutation P. Thus, any input permutation can be mapped to any of the possible $n!$ permutations. Finally, we can use the external-memory deterministic oblivious-sorting algorithm of Goodrich and Mitzenmacher [24], for instance, so as to use messages of size $m = O(n^{1/2})$, which will result in an algorithm that sorts in $O((n/m)\log_m(n/m)) = O((n/m))$ I/Os.

5.2.2 Oblivious Shuffle Model

In this section, we introduce a formal model for the oblivious shuffle of an array.

Definition 9 (Shuffle)**.** *A* shuffle $\mathcal{S}$ *is a pair of algorithms* $(\mathsf{Setup}, \mathsf{Shuffle})$*, as follows.*

- $(s, S) \leftarrow \mathsf{Setup}(1^k)$ *Given security parameter k, run the key generation algorithm for a symmetric encryption scheme* $(\mathsf{Enc}, \mathsf{Dec})$ *and store the key in secret state s. Also, allocate an auxiliary datastore S.*
- $(\mathsf{Enc}(\pi(A)), \alpha) \leftarrow \mathsf{Shuffle}(s, S, A, \pi)$ *Given secret state s, auxiliary data store S, an array input A, and a permutation π, return (1) the encryption of the permutation of A according to π; (2) a transcript α of the operations that transform* $\mathsf{Enc}(A)$ *to* $\mathsf{Enc}(\pi(\mathsf{A}))$ *using auxiliary space S.*

Transcript α is a sequence of l (request, response) pairs $\langle(r_1, g_1), \ldots, (r_l, g_l)\rangle$ that capture the evolution of the datastore via intermediate states $S_1, S_2, \ldots, S_{l+1}$. An invariant on each intermediate state is to store an encryption of some permutation *of A along with any auxiliary data. For example S_1 contains $\mathsf{Enc}(A)$ and S_l contains $\mathsf{Enc}(\pi(A))$. Setting $S_1 \leftarrow \{\mathsf{Enc}(A), S\}$, $s_0 \leftarrow s$, $g_0 \leftarrow \perp$ define the relationship between r_i and g_i as:*

$$\langle\ (s_i, r_i) \leftarrow \mathsf{GenRequest}(s_{i-1}, g_{i-1}),\quad (S_{i+1}, g_i) \leftarrow \mathsf{GenResponse}(S_i, r_i)\ \rangle.$$

Operations $\mathsf{GenRequest}$ and $\mathsf{GenResponse}$ generate a request r_i and a corresponding response g_i and are defined as follows:

- $(s_i, r_i) \leftarrow \mathsf{GenRequest}(s_{i-1}, g_{i-1})$ *Perform a computation based on a substructure of S_{i-1}, g_{i-1}, and generate next request to S_i, r_i.*
- $(S_{i+1}, g_i) \leftarrow \mathsf{GenResponse}(S_i, r_i)$ *Generate the response to request r_i on S_i: S_{i+1} is the datastore S_i updated according to r_i and g_i is the response to r_i with respect to S_i. For example, if r_i is a get request, then $S_{i+1} = S_i$ and g_i is the requested item. Also, if r_i is a put request, then g_i is empty.*

The private state s is updated if needed after every request.

In our cloud storage model, a shuffle $\mathcal{S}$ is a distributed computation executed by the user and the server. The user runs the Setup algorithm to generate the encryption key and requests the server to allocate some space. He then runs the Shuffle algorithm by accessing S through the server, that is, issuing requests to the server using GenRequest. The set of possible requests is defined by the storage model supported by the server. In our case this set is {get, put, getRange, putRange, getRangeDist, putRangeDist} (see Section 5.1.2). For every request r_i, the server executes $\mathsf{GenResponse}$, locally updating S for put requests and returning to the user the queried items for get requests.

5.2.3 Security of Oblivious Shuffle

We capture the security of a shuffle $\mathcal{S}$ against a curious server in the cloud storage model as a game, *Shuffle-IND*, between $\mathcal{S}$ and a probabilistic polynomial-time bounded (PPT) adversary $\mathcal{A}$. In this game, the inputs and outputs of $\mathcal{S}$ that are revealed to the server in the cloud storage model are also revealed to $\mathcal{A}$. However, the secret state s kept by the client, any updates to it and computations inside of $\mathsf{GenRequest}$ are kept private, since in the cloud model they are also hidden and happen on the user side.

The game starts with S running Setup once, allocating at the server space to be used in subsequent computations. $\mathcal{A}$ then tries to "learn" how $\mathcal{S}$ performs the shuffle on a sequence of m_1 input arrays and permutations picked by $\mathcal{A}$. Based on what $\mathcal{A}$ learns, she picks two challenges (A_0, τ_0) and (A_1, τ_1) each consisting of a data array to be permuted using a corresponding permutation. $\mathcal{S}$ secretly picks one pair and performs the shuffle according to it. The adversary is then allowed to observe $\mathcal{S}$ shuffling another sequence of m_2 (input, permutation) pairs, also picked by $\mathcal{A}$. Finally, $\mathcal{A}$ has to guess which challenge pair (input, permutation) $\mathcal{S}$ picked to shuffle. Note that at any time, $\mathcal{A}$ can ask $\mathcal{S}$ to perform a shuffle on any combination of A_0 or A_1 and permutations τ_0 or τ_1.

We now give a formal definition of the game.

Definition 10 (Shuffle-IND)**.** *Let A be an input array of size n picked by a PPT adversary $\mathcal{A}$. $\mathcal{A}$ and $\mathcal{S}$ engage in the following game.*

$\mathcal{S}$: $(s, S) \leftarrow \mathsf{Setup}(1^k)$.

for $j \in \{1, \ldots, m_1\}$, *where* m_1 *is* $\mathsf{poly}(k)$*:*

> $\mathcal{A}$*: Pick array* B_j *and a permutation* ρ_j.
>
> $\mathcal{S}$*: Execute* $(O_j, \alpha) \leftarrow \mathsf{Shuffle}(s, S, B_j, \rho_j)$. *Reveal* O_j *and* α *to* $\mathcal{A}$.

$\mathcal{A}$*: Pick* (A_0, τ_0) *and* (A_1, τ_1) *of the same length.*

$\mathcal{S}$*: Pick a secret bit* b *and execute* $(O, \alpha) \leftarrow \mathsf{Shuffle}(s, S, A_b, \tau_b)$. *Reveal* O *and* α *to* $\mathcal{A}$.

for $j \in \{m_1 + 1, \ldots, m_1 + m_2\}$, *where* m_2 *is* $\mathsf{poly}(k)$*:*

> $\mathcal{A}$*: Pick an encrypted array* B_j *and a permutation* ρ_j.
>
> $\mathcal{S}$*: Execute* $(O_j, \alpha) \leftarrow \mathsf{Shuffle}(s, S, B_j, \rho_j)$. *Reveal* O_j *and* α *to* $\mathcal{A}$.

$\mathcal{A}$*: Output bit* b'.

The adversary wins the game if $b = b'$.

Using the Shuffle-IND game, we now define an *oblivious shuffle.*

Definition 11 (Oblivious Shuffle). *Let* k *be the security parameter and* n *be a polynomial in* k. $\mathcal{S}$ *is an oblivious shuffle over* n *items if for every probabilistic adversary* $\mathcal{A}$ *running in time polynomial in* k*, the probability of winning the Shuffle-IND game,* $\Pr[b = b']$*, satisfies*

$$\Pr[b = b'] \leq \frac{1}{2} + \mathsf{negl}(k).$$

5.3 The Melbourne Shuffle Algorithm

In this section we develop an oblivious shuffle algorithm, whose goal is still to obliviously permute the collection, A, of n values, but with a simpler algorithm. Classic card shuffle methods (e.g., Knuth (or Fisher-Yates) [33], the riffle shuffle [3], Thorp shuffle [49]) are not data-oblivious, however, as anyone observing card swaps or riffles (interleaving two subdecks) of such methods can learn the final output permutation. are not data-oblivious, however, as anyone observing card swaps or riffles (interleaving two subdecks) of such methods can learn the final output permutation. To this end, we propose first data oblivious shuffle algorithm, the *Melbourne shuffle*. The Melbourne shuffle uses messages and private client memory of size $m = n^{1/c} \log n$, for any constant $c \geq 2$. We first explain and analyze $m = n^{1/2} \log n$ case, and later build a solution for the general case.

5.3.1 Intuition

The Melbourne shuffle uses private memory and messages of size $O(\sqrt{n} \log n)$, $O(n \log n)$ server storage and processes in $O(\sqrt{n})$ requests. An important ingredient of our solution is probabilistic encryption. Everything stored at the server is encrypted and every time an item is read from the server, the user decrypts it, re-encrypts it and writes it back. Since we use CPA-secure encryption, the ciphertexts produced for the same item always look different and, hence, the server, aka the adversary, cannot tell whether the ciphertexts correspond to the same item or not.

The goal of our oblivious shuffle is to reveal to the adversary only information that she would expect to see in a random permutation with very high probability. For example, even for a secret permutation picked uniformly at random, the adversary can guess with probability $1/n$ that the first element of the input array of size n appears in some location i of the output permutation. Continuing with this intuition, suppose we split the input array of size n into $\sqrt{n}$ buckets where every bucket has $\sqrt{n}$ items, and similarly for the output permutation. In this case, using the analysis of the balls-and-bins model, the adversary can guess that with high probability, each bucket in the output permutation has $O(\log n)$ elements from any particular bucket of the input array.

We build on the observation above and move elements from input buckets to output buckets by imitating the balls-and-bins process. That is, if the size of the input bucket is the same as the number of output buckets, we place $O(\log n)$ elements of every input bucket in every output bucket. If the number of elements in a bucket is much larger than the number of output buckets, i.e., the number of output buckets is $n^{1/c'}$ while input bucket has $n^{1/c}$ items, for constants c and c' s.t. $c' > c$, then we move $O(n^{1/c-1/c'})$ items to every output bucket.

The reader may have noticed that in the first example above, elements of an input bucket of size $\sqrt{n}$ are placed in $\sqrt{n}$ output buckets in batches of $O(\log n)$ items. What are the additional items? These additional items are referred to as *dummy items.* A dummy item is a real item with a fake key and some nonce value such that the size of the dummy and real item are equal. Moreover, since all the data is re-encrypted every time it is written to the server, the server cannot tell which items in a batch are real and which are dummy.

5.3.2 Overview

We assume that each element in the input array, A, is a key-value pair (x, v) for every $x \in D$. The algorithm has two phases: *distribution* and *clean-up*. For each phase, the data store S is split in several logical subparts: I, T and O. I is an array containing n encrypted items of the input A permuted according to some permutation π_0 (initially, π_0 is the identity). T is an encrypted temporary array used during the shuffle; after the shuffle is done, O contains the output, i.e., re-encrypted items of I permuted according to π. If the shuffle needs to be executed again, the user sets $I \leftarrow O$ and $\pi_0 \leftarrow \pi$. We further divide each subpart of S in buckets of equal size. The number of buckets and how it effects the runtime of the algorithm will be determined later.

During the distribution phase items of every bucket of I along with some dummy items are re-encrypted and distributed equally among buckets of T. Here, the distribution of item (x, v) is done according to its final location $\pi(x)$ in O. After the distribution phase the intermediate array T contains real and dummy items. Moreover, the items appear in correct buckets but not in correct positions within each bucket. The clean-up phase remedies this by reading one bucket at a time, removing dummy items, distributing the real items correctly within the bucket and writing the bucket to O.

The distribution phase alone cannot produce every possible permutation since the number of items sent from a bucket of I to a bucket of T is limited. E.g., the identity permutation cannot be achieved. To rectify this, we execute two shuffle passes. First, for a permutation π_1 picked uniformly at random and then for the desired permutation π. Although this framework still allows failures, our algorithm can produce every permutation, failing with very small probability independent of the desired permutation π.

5.3.3 Algorithm

The complete shuffle algorithm $\mathsf{shuffle}(I, \pi, O)$ is shown in Algorithm 4 where I is the encryption of the input array A, π is the desired permutation and the last argument is the output array where the algorithm is expected to put an encryption of $\pi(A)$. We omit the Setup from the discussion since it is trivial: the client simply runs a key generation to setup a secure encryption scheme and a seed generator for pseudo random permutations.

The algorithm makes two calls to $\mathsf{shuffle\_pass}$ (Algorithm 5), first for a random permutation π_1 and then for the desired permutation π. We proceed with the description of $\mathsf{shuffle\_pass}(I, T, \rho, O)$ where I and O are defined as in $\mathsf{shuffle}$, T is a temporary array and ρ is the desired permutation. We use the convention of giving arrays I, T and O as inputs to the shuffle pass algorithm for the ease of explanation. In the cloud storage scenario that we consider here, one simply specifies the location where these arrays are stored remotely, e.g., the name of a file and a location within it. Given an input array of size n, this method has messages and client's private memory of size $O(\sqrt{n} \log n)$ and server memory of size $O(n \log n)$. These user and server memory requirements are temporary and are reduced to $O(1)$ and n, respectively, when the shuffle is finished. As mentioned before, method $\mathsf{shuffle\_pass}$ is split into a distribution phase and a clean-up phase.

Algorithm 4 The complete Melbourne shuffle algorithm, shuffle(I, π, O), where the user can read and store in private memory M up to $\sqrt{n} \times p \log n$ elements, $p \geq e$.

$\boldsymbol{I}$: array of n encrypted elements (x, v); $\boldsymbol{\pi}$: permutation; $\boldsymbol{O}$: permutation of I according to π, where every element is re-encrypted.

```
1: Let $\pi_1$ be a random permutation
2: Let $T$ be an empty array of size $n \times p \log n$ stored remotely
3: shuffle_pass$(I, T, \pi_1, O)$
4: $I \leftarrow O$
5: shuffle_pass$(I, T, \pi, O)$
```

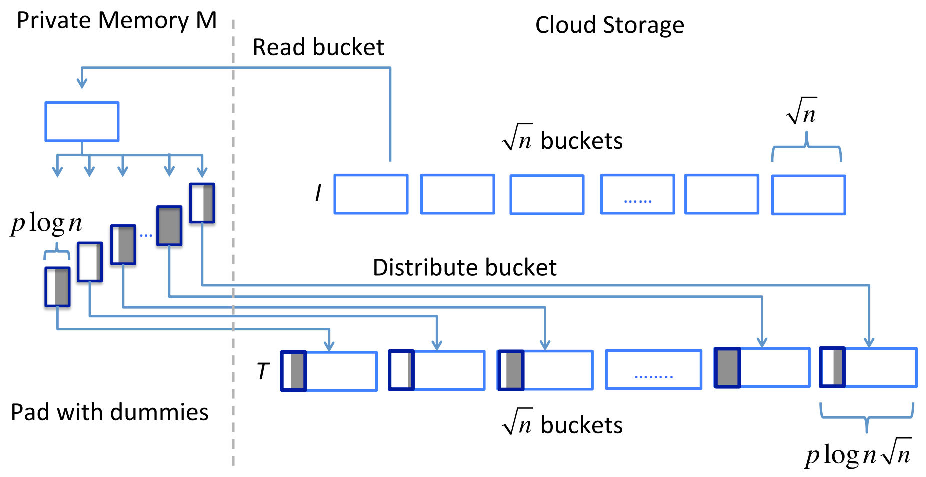


Figure 5.1: Illustration of the distribution phase of shuffle_pass (Algorithm 5). Shadowed regions represent dummy values added to pad each batch to the size of $p \log n$. The batches are encrypted, hence, one cannot tell where and how many dummy values there are in each batch.

Distribution Phase The distribution phase of method shuffle_pass (Algorithm 5), shown in Figure 5.1, imitates throwing balls into bins by putting elements from every bucket of I to every bucket of T according to the permutation ρ. In particular, a batch of $p \log n$ encrypted elements from every bucket of I is put in every bucket of T (rev_bucket[id$_T$] in the pseudo-code). Here, p is a constant and is determined in the analysis.

Each batch contains real and dummy elements. The first batch is filled in with real elements (x, v) that would go to the first bucket in O according to ρ, i.e.,

Algorithm 5 Single pass $\mathsf{shuffle\_pass}(I, T, \rho, O)$ of the Melbourne shuffle algorithm, where the user can read and store in private memory M upto $\sqrt{n} \times p \log n$ elements.

I: array of n encrypted elements (x, v); ***T***: auxiliary array that fits $n \times p \log n$ encrypted elements, where p is a constant that is a parameter of the algorithm; $\boldsymbol{\rho}$: permutation; ***O***: permutation of I according to ρ, where every element is re-encrypted.

```
1: $\mathsf{max\_elems} \leftarrow p \log n$
2: $\mathsf{num\_buckets} \leftarrow \sqrt{n}$
3: $\{$*Distribution phase: distribute elements of $I$ into $T$*$\}$
4: **for** $\mathsf{id}_I \in \{0, \ldots, \mathsf{num\_buckets} - 1\}$ **do** $\{$read buckets of $I\}$
5:     $\mathsf{bucket}_M \leftarrow \mathsf{getRange}(I, \mathsf{id}_I \times \sqrt{n}, \sqrt{n})$
6:     $\mathsf{rev\_bucket}_M \leftarrow \mathsf{empty\_map}()$ $\{$Reverse map of bucket ids in $T$ to elements$\}$
7:     **for** $e \in \mathsf{bucket}_M$ **do** $\{$Assign elements their bucket ids in $T\}$
8:         $(x, v) \leftarrow \mathsf{Dec}(e)$
9:         $\mathsf{id}_T \leftarrow \lfloor \rho(x)/\sqrt{n} \rfloor$ $\{$Bucket id of element $(x, v)$ in $T$ according to its location
           in $O\}$
10:        $\mathsf{rev\_bucket}_M[\mathsf{id}_T].\mathsf{add}(\mathsf{Enc}(x, v))$ $\{$Collect elements of same bucket$\}$
11:    **end for**
12:    $\{$Can be done via a single $\mathsf{putRangeDist}$ for $\sqrt{n}$ batches of size $\mathsf{max\_elems}\}$
13:    **for** $\mathsf{id}_T \in \{0, \ldots, \mathsf{num\_buckets} - 1\}$ **do** $\{$Distribute $\mathsf{bucket}_M$ in buckets of $T\}$
14:        **if** $\mathsf{size}(\mathsf{rev\_bucket}_M[\mathsf{id}_T]) > \mathsf{max\_elems}$ **then**
15:            **fail** $\{\rho$ moves more than $p \log n$ elements from a bucket of $I$ to a bucket
               of $T\}$
16:        **end if**
17:        $\{$Hide how many real elements go to $T$'s buckets by padding with encrypted
           dummies$\}$
18:        $\mathsf{rev\_bucket}_M[\mathsf{id}_T] \leftarrow \mathsf{dummy\_pad}(\mathsf{rev\_bucket}_M[\mathsf{id}_T], \mathsf{max\_elems})$
19:        $\{$Write a batch of $\mathsf{max\_elems}$ from every bucket of $I$ to every bucket of $T\}$
20:        $\mathsf{putRange}(T, \mathsf{id}_T \times \sqrt{n} \times \mathsf{max\_elems} + \mathsf{max\_elems} \times \mathsf{id}_I, \mathsf{rev\_bucket}_M[\mathsf{id}_T])$
21:    **end for**
22: **end for**
23: $\{$*Clean-up phase: clean $T$ and write the result to $O$*$\}$
24: **for** $\mathsf{id}_T \in \{0, \ldots, \mathsf{num\_buckets} - 1\}$ **do** $\{$read buckets of $T\}$
25:    $\mathsf{bucket}_M \leftarrow \mathsf{getRange}(T, \mathsf{id}_T \times \sqrt{n} \times \mathsf{max\_elems}, \sqrt{n} \times \mathsf{max\_elems})$
26:    $\{$Decrypt the bucket, remove $\mathsf{dummy}$, sort real elements using $\rho$ and re-
       encrypt$\}$
27:    $\mathsf{bucket}_M \leftarrow \mathsf{clean}(\mathsf{bucket}_M)$
28:    $\{$The distribution phase guarantees that $\mathsf{bucket}_M$ contains exactly $\sqrt{n}$ ele-
       ments$\}$
29:    $\mathsf{putRange}(O, \mathsf{id}_T \times \sqrt{n}, \mathsf{bucket}_M)$
30: **end for**
```

the elements for which $\lfloor \rho(x)/\sqrt{n} \rfloor = 0$. Similarly for every other batch. Since a bucket of I contains only $\sqrt{n}$ elements and we put $\sqrt{n} \times p \log n$ elements in total in all buckets in T, most batches will have less than $p \log n$ elements. We pad such batches with dummy elements to hide where and how many elements of I's bucket are placed in T (line 18). Note that a batch is re-encrypted before it is written to T, completely hiding the content and making it impossible to recognize where dummy or real elements are (lines 10 and 18). If according to ρ more than $p \log n$ elements are mapped from a bucket of I to a bucket of T, the algorithm fails (line 15). We later consider what happens in case of a failure. We note that $\sqrt{n}$ calls to putRange in the loop in lines 13-21 is for the ease of explanation only. These calls can be substituted by a single call putRangeDist, putting $\sqrt{n} \times p \log n$ elements all at once. Hence, for every bucket read from I, there is only *one* corresponding write to T.

Clean-up Phase The distribution phase leaves T with two problems: first, though the elements are in correct buckets according to ρ they are not in the correct locations inside the buckets, and second, T contains dummy elements. To remedy these problems, the clean-up phase in Algorithm 5, illustrated in Figure 5.2, proceeds by reading buckets of T of size $\sqrt{n} \times p \log n$ and writing in their place buckets of size $\sqrt{n}$.

When processing each bucket, the algorithm removes dummy elements, sorts the remaining content of every bucket according to their final location in O (line 27). It is important to note that each written bucket contains exactly $\sqrt{n}$ elements before it is being written back. This follows from the fact that elements were distributed to buckets according to the permutation ρ and the algorithm failed in the distribution phase for those ρ that would have resulted in more than $\sqrt{n}$ elements in each bucket.

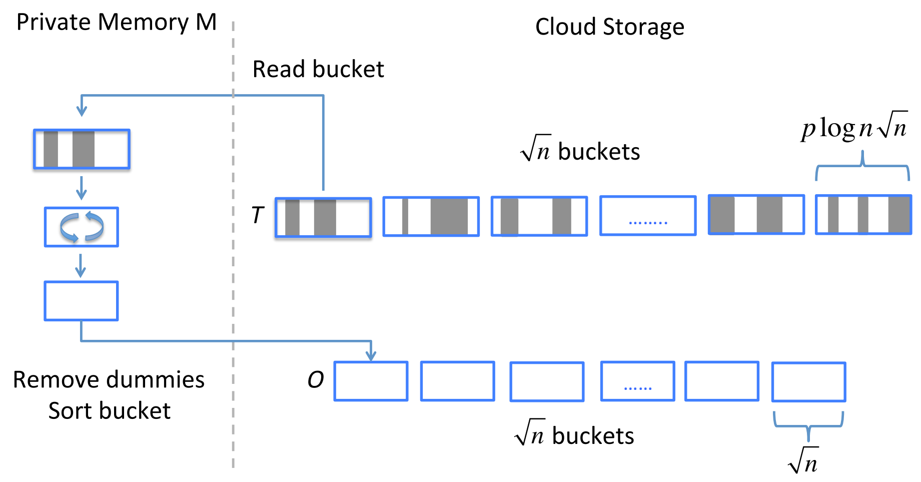


Figure 5.2: Illustration of the clean-up phase of shuffle_pass Algorithm 5). Shadowed regions represent dummy values that are removed during the clean-up phase.

Performance The performance of the Melbourne shuffle is summarized in the following theorem.

Theorem 12. *Given an input array of size n, the Melbourne shuffle (Algorithm 4) executes $O(\sqrt{n})$ operations, each exchanging a message of size $O(\sqrt{n} \log n)$, between a user with private memory of size $O(\sqrt{n} \log n)$ and a server with storage of size $O(n \log n)$. Also, the user and server perform $O(n \log n)$ work.*

Proof. We first note that $\sqrt{n}$ calls to putRange in the loop in lines 13-21 is for the ease of explanation only. These calls can be substituted by a single call putRangeDist, putting $\sqrt{n} \times p \log n$ elements all at once.

A single shuffle pass in Algorithm 5 requires $2\sqrt{n}$ calls to getRange, $\sqrt{n}$ calls to putRangeDist, and $\sqrt{n}$ calls to putRange, assuming the user and the server can exchange up to $\sqrt{n} \times p \log n$ elements in a single request. The shuffle in Algorithm 4 requires $8\sqrt{n}$ requests in total since it makes 2 calls to the shuffle pass procedure. The private memory required at the user to perform the shuffle is $\sqrt{n} \times p \log n$. The required server's memory is $n \times p \log n$. However, this overhead is temporary since the increase in memory happens only during the shuffle pass and is reduced to n when the shuffle is finished. Similarly for the user, the memory of size $\sqrt{n} \times p \log n$ is required only during the shuffle. We note that the total computation for the user and the server is $O(n \log n)$. $\square$

5.3.4 Security Analysis

In this section, we show that the Melbourne shuffle (Algorithm 4) is oblivious for every permutation π with high probability.

Definition 13. *Let A be an array of n elements such that every $x \in [1, n]$ is at location $\pi_0(x)$ in A. Let B be an array that stores a permutation π of elements in A, i.e., $B = \pi(A)$. Split A and B in $\sqrt{n}$ buckets of equal size and fix a constant $p \geq \mathrm{e}$. Let π be a permutation on n elements where every bucket of B contains at most $p \log n$ elements of every bucket of A. We refer to the set of all such permutations as $P(\pi_0)$.*

Lemma 14. *The size of set $P(\pi_0)$ is $(1 - \mathsf{negl}(n)) \times n!$, for every permutation π_0.*

Proof. Let π be a random permutation from all possible $n!$ permutations. We consider the relationship between the input array A and a permutation of A, $B = \pi(A)$. We start by splitting A and B in buckets of size $\sqrt{n}$ and numbering buckets from 1 to $\sqrt{n}$ using their order in each array. The analysis below estimates how many permutations can be constructed by restricting the maximum number of elements from a bucket of A appearing in any bucket of B to $p \log n$.

Let X_a^b be a random variable that measures the number of elements from ath bucket of A present in bth bucket of B. The mean value of X_a^b is 1, since we are distributing $\sqrt{n}$ elements of a among $\sqrt{n}$ buckets of B. Although X_a^b, for $1 \leq a, b \leq \sqrt{n}$, variables are dependent between each other, we can use the Poisson Approximation [39, Chapter 5.4] and instead work with n independent Poisson random variables Y_a^b with mean 1.

Given n variables Y_a^b we are interested in bounding the probability of the event that there is no a and b such that $Y_a^b \geq p \log n$. For a specific a and b it is:

$$\Pr[Y_a^b \geq p \log n] \leq \frac{1}{\mathrm{e}} \left(\frac{\mathrm{e}}{p \log n} \right)^{p \log n}.$$

Using union bound, the probability that at least one of the Y_a^bs is greater than $p \log n$ is at most

$$n \frac{1}{\mathrm{e}} \left(\frac{\mathrm{e}}{p \log n} \right)^{p \log n}.$$

Since Y_a^bs are a Poisson approximation of the variables X_a^b, the probability that at least one of the X_a^bs is greater than $p \log n$ is at most

$$2n \frac{1}{\mathrm{e}} \left(\frac{\mathrm{e}}{p \log n} \right)^{p \log n}.$$

Setting $p \geq \mathrm{e}$ we get

$$2n \frac{1}{\mathrm{e}} \left(\frac{\mathrm{e}}{p \log n} \right)^{p \log n} \leq \frac{2n}{(\log n)^{p \log n}} = \frac{2n}{n^{p \log \log n}} = 2\mathsf{negl}(n) = \mathsf{negl}(n).$$

□

Lemma 15. *Let π_0 be the initial permutation of n elements in the input array I. Method* shuffle_pass *(Algorithm 5) succeeds for all permutations $\rho \in P(\pi_0)$.*

Proof. The algorithm allocates elements of I in O according to ρ by first putting them into corrects buckets (lines 4–22) and then sorting every bucket using ρ (line 27). By construction the algorithm fails for any permutation ρ that requires more than $p \log n$ elements from a bucket of I mapped to buckets of O (line 14–16). □

Lemma 16. *Method* shuffle(I, π, O) *(Algorithm 4) is a randomized shuffle algorithm that succeeds with very high probability.*

Proof. Let π_0 be the initial permutation of the input array I. Algorithm 4 makes two calls to shuffle pass which succeeds for all possible permutations except for $1/n^{\Omega(\log \log n)}$ fraction of them (Lemmas 14 and 15). The first shuffle pass is executed for permutation π_1 on an input permuted according to some permutation π_0. Since π_1 is picked using internal random coins, the probability of the shuffle pass failing is independent of π_0 and is bounded by $1/n^{\Omega(\log \log n)}$. If the first shuffle pass did not fail, the shuffle pass is executed second time with input permutation π. The second shuffle is executed on the input array that is permuted according to a random permutation $\pi_1 \in P(\pi_0)$. The second pass does not fail iff $\pi_1 \in P(\pi)$. Hence, Algorithm 4 fails if $\pi_1 \notin P(\pi_0)$ or $\pi_1 \notin P(\pi)$. By Lemma 14, the probability of either of these events is negligible in n, hence the basic Melbourne shuffle succeeds with very high probability for any π_0 and π. □

We show that method shuffle_pass (Algorithm 5) is oblivious by mapping it to the Oblivious Shuffle Model in Section 7.2, extracting the corresponding transcript and showing that the transcript reveals no information about the underlying permutation if the encryption scheme is CPA secure (see Section 3.3).

Method shuffle_pass (Algorithm 5) corresponds to GenRequest. Calls to getRange and putRange trigger calls to GenResponse at the server. We do not describe GenResponse since it depends on the implementation of the storage provider. We are only interested in the fact that it uses server's state S to store and maintain arrays I, T and O. The transcript α of the shuffle execution is defined as follows. The request r_i is either $\mathsf{getRange}(S, i, l)$ or $\mathsf{putRange}(S, i, a)$, e.g., in line 29 (S, x, a) is $(O, \mathsf{id}_T \times \sqrt{n}, \mathsf{bucket}_M)$. The response g_i to getRange is an array a, e.g., a contains $\sqrt{n}$ elements in line 5 and is stored in bucket_M. The response to putRange is empty. We first analyze the metadata (Definition 8) that corresponds to every request between the client and the server, and show that, unless the algorithm fails, they depend on the size of the input only, and are independent from the input array and the desired permutation. Hence, we obtain that the Melbourne shuffle is a data independent shuffle algorithm. We finally show that if the content exchanged is encrypted, as it is in method shuffle_pass (Algorithm 5), the Melbourne shuffle (Algorithm 4) is oblivious.

Lemma 17. *The metadata of requests exchanged between the client and the server in method* shuffle_pass *(Algorithm 5) is independent of permutation π_0 of the input I and output permutation $\rho \in P(\pi_0)$, and depends only on n.*

Proof. Let α be a sequence of (request, response) pairs exchanged between the client and the server, denoted as (r_i, g_i), where r_i is either a getRange or putRange and g_i is bucket_M for getRange and empty for getRange. The sequence α can be further split in $\sqrt{n}$ $(\mathsf{getRange}, \mathsf{putRange})$ calls that correspond to distribution phase and $\sqrt{n}$ $(\mathsf{getRange}, \mathsf{putRange})$ during the clean-up phase.

The metadata of putRange is the name of the array, the location within the array and how many elements should be read. In the algorithm these correspond to reading an array of size n sequentially in buckets of size $\sqrt{n}$ (distribution phase, line 5) and $\sqrt{n} \times p \log n$ (clean-up phase, line 25). These data depend only on n and p. The metadata for a putRange call consists of the array to be accessed, the location where to put the data and the size of the data to be written. In the algorithm, first calls to putRange place $\sqrt{n}$ batches of size $p \log n$ in the temporary array to locations that depend on the input bucket that has been read using getRange (line 20). These locations are deterministic since getRange simply scans the input array. The second sequence of calls to putRange happens during the clean-up phase when buckets of size $\sqrt{n}$ are written sequentially to the output array (line 29). These calls are also deterministic. It is also easy to show that the transcript where a sequence of $\sqrt{n}$ calls to putRange in lines 13-21 is substituted with a single call to putRangeDist is also deterministic. □

Lemma 18. *Let π_0 be the initial permutation of n elements in the input array I and let ρ be a permutation from the set $P(\pi_0)$. Method* shuffle_pass(I, T, ρ, O) *(Algorithm 5) is an oblivious shuffle according to Definition 11.*

Proof. We showed in Lemma 15 that Algorithm 5 succeeds for all $\rho \in P(\pi_0)$. We also showed that all metadata in the transcript that is revealed to the adversary $\mathcal{A}$ in the Shuffle-IND game in Definition 10 after every call to Shuffle is independent of data content and hence can be determined based only on n and is the same for any choice of input and output. The data content exchanged in each call does depend on the data, however, it is always encrypted. We show that the security of the shuffle depends on the security of the underlying encryption scheme.

To the contrary, we assume that there is a PPT adversary $\mathcal{A}$ that can distinguish with a non-negligible advantage two permutations τ_0 and τ_1 by observing the transcript of one of them. Hence, this adversary can win Shuffle-IND game. We show that if $\mathcal{A}$ exists then we can construct an adversary $\mathcal{B}$ who can use $\mathcal{A}$ to win Enc-IND-CPA game with a non-negligible advantage, which would break our assumption about the encryption scheme. We recall that in Enc-IND-CPA game $\mathcal{B}$ has access to oracles Enc and Dec and can encrypt and decrypt sequences of messages of his choice, except asking for the decryption of the challenge ciphertext.

We construct the adversary $\mathcal{B}$ as follows. $\mathcal{B}$ does not need to run Setup since he has oracle access to Enc and Dec. $\mathcal{A}$ starts making calls to Shuffle on chosen pairs of input arrays and permutations. $\mathcal{B}$ imitates the shuffle by responding with the encrypted permutation and a transcript α, that he can produce himself. He continues doing so until $\mathcal{A}$ comes up with a challenge of two (input, permutation) pairs (A_0, τ_0) and (A_1, τ_1). $\mathcal{B}$ first creates a static transcript that will be the same for both permutations. He then, extracts all the calls to be made to Enc into two sequences: one that corresponds to (A_0, τ_0) and one to (A_1, τ_1). He gives these two sequences of elements to be encrypted to his own challenger. The challenger picks one sequence at random, encrypts all its plaintexts (i.e., elements) one by one and gives the result to $\mathcal{B}$. $\mathcal{B}$ combines the ciphertexts with the metadata, that depend only on n, to create a valid transcript α and sends it to $\mathcal{A}$. He continues, responding to Shuffle requests from $\mathcal{A}$ until $\mathcal{A}$ outputs his guess b for the pair (A_b, τ_b). $\mathcal{B}$ outputs b as his guess for which sequence of messages his challenger picked. Since $\mathcal{A}$'s advantage in winning the game is non-negligible so is $\mathcal{B}$'s. $\square$

Theorem 19. *The Melbourne Shuffle (Algorithm 4) is a randomized shuffle algorithm that succeeds with very high probability and is data-oblivious according to Definition 11.*

Proof. Algorithm 4 makes two calls to method shuffle_pass, which is oblivious by Lemma 18. If Algorithm 4 is not oblivious, then there is a PPT adversary $\mathcal{A}$ who can distinguish shuffle of two permutations. If $\mathcal{A}$ exists, we can build an adversary $\mathcal{B}$ who can break the security of the underlying shuffle_pass using $\mathcal{A}$. Whenever $\mathcal{A}$ makes a call to Shuffle for a permutation π, $\mathcal{B}$ first picks a random permutation π_1

and calls shuffle_pass on π_1. It then uses the output of this call and π to make another call to shuffle_pass, this time returning the output to $\mathcal{A}$. This continues until $\mathcal{A}$ comes up with two challenge pairs (input, permutation) (A_0, τ_0) and (A_1, τ_1). $\mathcal{B}$ first picks a random permutation and runs shuffle-pass on it. He then gives his own challenger the output of this shuffle along with (A_0, τ_0) and (A_1, τ_1). The challenger returns to $\mathcal{B}$ the shuffle according to (A_b, τ_b), keeping b secret. $\mathcal{B}$ forwards what he receives to $\mathcal{A}$. $\mathcal{B}$ continues replying shuffle requests of $\mathcal{A}$ as he did before until $\mathcal{A}$ makes a guess for b. $\mathcal{B}$ outputs this guess as his own. $\square$

Note that method shuffle_pass outputs fail for some permutations. Whenever it does so, $\mathcal{B}$ sends this information to $\mathcal{A}$. However, as showed in Lemma 16 this happens with negligible probability for any pair of input and output permutations and reveals nothing to the adversary about the output permutation since the failure is due to secret random bits.

5.3.5 The Melbourne Shuffle with Small Messages

The Melbourne shuffle can be extended to work with messages and private memory of size $n^{1/c} \log n$, for $c \geq 3$.

The idea behind the approach is to run the algorithm recursively with depth $c-1$. For a fixed c, one first splits the output in large buckets of size $n^{(c-1)/c}$ and executes the shuffle as in the square root case: distributing $n^{1/c}$ among $n^{1/c}$ large buckets. We call this the first level of the recursion. After this, each large bucket has correct elements but not in correct buckets nor positions. The square root shuffle is executed again on each large bucket, but now splitting the large bucket of size $O(n^{(c-1)/c} \log n)$ in $n^{(c-2)/c}$ buckets and again using only $n^{1/c}$ private memory. The client follows the recursion until the size of the inner buckets becomes $O(n^{2/c})$ when elements can be distributed in their correct buckets of size $n^{1/c}$. At this point, the buckets are small enough that they can be read to private memory during the clean-up phase and be placed in correct positions within their bucket. Hence, there are $c-1$ levels of recursion. Each level i requires a block of $n^{(c-i)/c}$ buckets, each of size $n^{1/c}$, to be distributed among $n^{1/c}$ output buckets. Since every level has $n^{(i-1)/c}$ blocks, the square root solution is required to be executed $n^{(i-1)/c}$ times per level, each making $O(n^{(c-i)/c})$ accesses. Hence, the total number of requests can be expressed as $\sum_{i=1}^{c-1} O(n^{(c-i)/c} \times n^{(i-1)/c}) = O((c-1)n^{(c-1)/c})$.

The Melbourne shuffle uses private memory and messages with a multiplicative $\log n$ factor. Hence, a naive extension of this algorithm to small messages could accumulate the $O((\log n)^c)$ factor. This happens due to reading $n^{1/c}$ elements and writing back $O(n^{1/c} \log n)$ elements and clean-up phase not being able to reduce this to $n^{1/c}$ until the last level of the recursion. One can prevent this by observing that when distributing $n^{2/c}$ elements among $n^{1/c}$ output buckets, every bucket will have at most $n^{1/c}$ of its elements in every output bucket, with very high probability (by using Chernoff bounds [39, Chapter 4]). Hence, after distributing $O(\log n)$ elements from buckets of size $n^{1/c}$, we can make another sequential pass, reading $n^{1/c} \log n$ elements at a time (i.e., the elements that were contributed by batches of size $\log n$ from $n^{1/c}$ buckets) and applying the fact that all together they could not contribute more than $O(n^{1/c})$ elements, and hence writing back only $O(n^{1/c})$. Note that we remain data-oblivious, since again we are using the observation of what the adversary would expect to see with very high probability.

We note that each recursive level does not depend on higher levels and, hence, has the same memory requirements as a single shuffle pass of the corresponding algorithm.

Theorem 20. *Given an integer constant $c \geq 3$ and an input array of size n, the Melbourne shuffle executes $O(cn^{(c-1)/c})$ operations, each exchanging a message of size $O(n^{1/c} \log n)$ between a user with private memory of size $O(n^{1/c} \log n)$ and a server with storage of size $O(n \log n)$. Also, the user and server perform $O(cn \log n)$ work.*

5.4 The Square-Root Based OS Solution

In this section, we give an overview of a secure and efficient oblivious storage scheme that uses the Melbourne shuffle. The oblivious storage (OS) we consider here follows the framework proposed in [21] (see Section 3.7 for the description). Here, we assume that the user and the server can exchange messages of size $O(\sqrt{n} \log n)$ and the client's memory is also $O(\sqrt{n} \log n)$. We change the solution of [21] as follows.

Setup This phase follows the description in Section 3.7 until one the shuffle of I is required. The client picks a secret permutation, PRP π and calls the Melbourne shuffle via $\mathsf{shuffle}(I, \pi, O)$ where O is the location at the server where the shuffled and encrypted array of A is stored. We set $I \leftarrow O$ for the access phase. Similar to the original square-root solution, the user also allocates at the server an empty cache C that can fit encryptions of up to $\sqrt{n}$ requested elements. Recall that the number of non-empty locations in C is the total number of requests that were made to remote storage since the last time I was shuffled. We refer to how many elements are present in C as l. As before, the client remembers the seed that generated π so that he can find the elements during the access phase and the encryption key in his private memory.

Access Phase Access phase also follows the description in Algorithm 2 but request the cache C with a single request since the cache fits in one request message and in client's memory. It decrypts the cache and checks if an element with the key x is present in C. If so, it remembers the element (x, v). Then, a location in I is accessed as follow. If the element was found in the cache a fake element with the key $n + l$ is accessed by requesting the location $\pi(n + l)$ of I. Otherwise, the location that stores the element with the key x is accessed, by requesting location $\pi(x)$ of I. After reading the cache and making one request to I, the user has the desired element (x, v). If the original request was read he writes encrypted element (x, v) to the first empty location in C on the server, if it was a write, the client writes (x, v') instead. This phase can proceed this way for $\sqrt{n} - 1$ more requests, after that the cache fills up and the rebuild phase follows. This phase requires 3 accesses to the remote storage per every requested element.

Rebuild Phase Recall that the goal of the rebuild phase is to free C by placing updated elements back to I and shuffle I using a new secret permutation π' that is independent of π. This step has to be done in a data-oblivious manner to prevent correlations between access patterns to I before and after reshuffle. One first writes C to I by reading C in private memory and then reading buckets of size $\sqrt{n}$ of I, updating them with elements of C, if needed, and writing them back re-encrypted. After merging the cache C and I we are ready to call the Melbourne shuffle via shuffle(I, π', O). (Recall that in [21] this step was made using oblivious sorting algorithms such as Batcher's sorting network or AKS making $O(n \log^2 n)$ or $O(n \log n)$ requests, respectively). The client updates the seed that generated π' in his private memory and allocates an empty cache C at the server. The rebuild takes $O(\sqrt{n})$ accesses if we use the Melbourne Shuffle. Since the shuffle happens after $\sqrt{n}$

accesses we can amortize the rebuild overhead over the $\sqrt{n}$ elements that caused it achieving $O(1)$ amortized overhead for every element.

The performance of the above OS scheme based on the Melbourne shuffle is summarized in the following theorem.

Theorem 21. *The randomized oblivious storage scheme based on the Melbourne shuffle has the following properties, where n is the size of the outsourced dataset:*

- *The private memory at the client and each message exchanged between the client and server have size $O(\sqrt{n}\log n)$.*
- *The memory at the server has size $O(n\log n)$ during the rebuild phase and $O(n)$ during the access phase.*
- *The amortized access overhead to perform a storage request is $O(1)$.*

In Chapter 6 we show that we can perform access and rebuild in parallel and deamortize the rebuild overhead to achieve the worst case overhead of $O(1)$.

5.5 The cth-Root Based OS Solution

We apply recursion to the square root solution to support messages and private user memory of size $n^{1/c}\log n$. This solution uses a cache of size $n^{1/c}$, which fits into private memory, and $c-1$ levels of additional storage. Each level i is large enough to contain $n^{(i+1)/c}$ real elements and $n^{i/c}$ fake elements. The cache and levels 1 to $i-1$ have a similar cache functionality for level i as the cache C in the square root solution, except they can store together $O(n^{i/c})$ previously accessed elements.

Each level $i < c - 1$ contains $n^{(i+1)/c} + n^{i/c}$ buckets of size $O(\log n)$ that allows one to store $n^{(i+1)/c} + n^{i/c}$ elements in a hash table and avoid collisions with very high probability (see Section 3.4). Buckets with fewer than $\log n$ elements, real or fake, are filled in with dummies. The last level, level $c - 1$, has n real and $n^{(c-1)/c}$ fake elements, hence a permutation can be used to store and access the elements. Given the above memory arrangement at the server, the access and rebuild phases proceed as follows.

Access phase requires a total of $c + 1$ accesses to the server: read the cache of size $n^{1/c}$, read $O(\log n)$ size bucket from every $c - 2$ levels, read one element from the last level, write an updated cache back. Note that each bucket can be read into memory since we assume private memory and messages of size $O(n^{1/c} \log n)$. Hence, $O(c)$ accesses are required to access an element obliviously. The access phase proceeds as follows: read the cache, if an element is found access a new fake element from level 1, and proceed with fake accesses from then on. Otherwise, look up the bucket using a hash function of level 1 where the element is supposed to be if it was read before. Again, proceed with consequent fake accesses, if the element was found, or keep looking for the element.

Rebuild phase After $n^{1/c}$ elements have been accessed, level 1 is rebuilt taking $O(n^{1/c} \log n)$ accesses to be rebuilt using the Melbourne Shuffle for small messages (see Section 5.3.5). Similarly, when (i-1)th level is full it requires shuffling of $n^{(i+1)/c} \log n$ elements at level i using $O(i \times n^{i/c} \log n)$ accesses. Finally, the last level requires $O((c - 1)n^{(c-1)/c})$ accesses to rebuild. We could amortize the cost of the rebuilds to get $O(c \log n)$ amortized, access overhead per every element.

Though, this solution resembles the Hierarchical Solution of [21] by using buckets of size $\log n$ at every level, it has two important differences. The expansion factor from level to level is $n^{1/c}$, instead of 2, hence, we only have c levels, and the rebuild phase uses our shuffle algorithm instead of a more expensive oblivious sort.

Theorem 22. *The randomized oblivious storage scheme based on the Melbourne shuffle with small messages has the following properties, where n is the size of the outsourced dataset and c is a constant such that $c \geq 3$:*

- *The private memory at the client and each message exchanged between the client and server have size $O(\sqrt[c]{n} \log n)$.*
- *The memory at the server has size $O(n \log n)$ during the rebuild phase and $O(n)$ during the access phase.*
- *The amortized access overhead to perform a storage request is $O(c \log n)$.*

In the next Chapter we show that we can perform access and rebuild in parallel and deamortize the rebuild overhead to achieve the worst case overhead of $O(c \log n)$.

Chapter Six

Deamortization of Oblivious Access Schemes

The oblivious RAM solutions that we have presented in Sections 4 and 5, including some of the solutions in Table 2.1, try to minimize the amortized access overhead of each simulated request. However, depending on when the request occurs the user may have to wait for as many as $\Omega(n)$ accesses till she can proceed with her requests. In this chapter we deamortize solutions [21] (Section 3.7) and Section 4 such that each request takes a sub-linear worst-case bounded amount of time. The technique for [21] can be trivially extended to the solution in Section 5 where we use the Melbourne Shuffle instead of oblivious sorting during the rebuild phase. The technique presented here can be also used to deamortize other hierarchical-based ORAM solutions [50, 34].

6.1 The Square-Root Scheme

We first present an oblivious RAM simulation method with $O(\sqrt{n}\log^2 n)$ access overhead in the worst case to demonstrate some of the ideas behind the more efficient technique developed in Section 6.2. The method is based on the square-root approach originally proposed in [21], which has $O(n\log^2 n)$ worst-case access overhead and $O(\sqrt{n}\log^2 n)$ amortized access overhead (see Section 3.7).

Intuition The most expensive step of the square-root approach is building a new table, where items and dummy values are ordered using a new pseudo-random permutation. This step is executed every $\sqrt{n}$ requests and takes $O(n\log^2 n)$ accesses. Our idea is to split the accesses for the rebuild into $\sqrt{n}$ batches, each with $O(\sqrt{n}\log^2 n)$ accesses, and to execute each batch after processing a request so that the new table is ready to be used after processing $\sqrt{n}$ requests. We will show how this idea can be

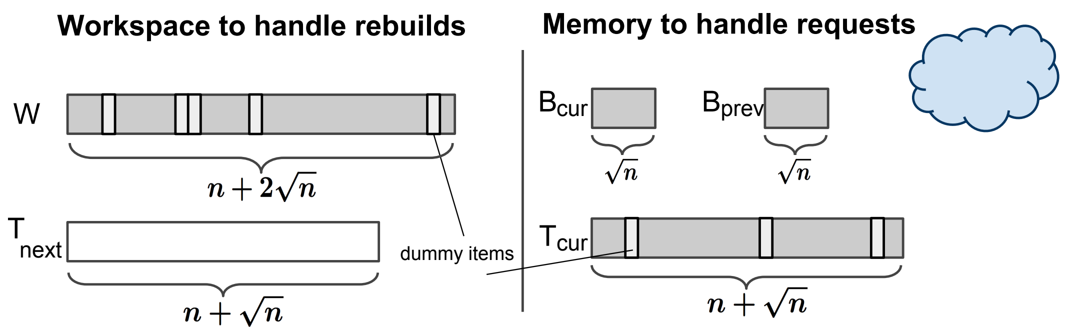


Figure 6.1: Memory layout of the data repository during oblivious RAM simulation of our deamortized version of the square-root solution (Section 6.1).

implemented while preserving obliviousness and keeping the same asymptotic access overhead and storage overhead as the original method.

6.1.1 Setup

Memory Layout We organize the memory on the data repository into five areas. We make use of two buffers, B_{cur} and B_{prev}, each of size $\sqrt{n}$. We also have two tables, T_{cur} and T_{next}, each of size $n + \sqrt{n}$. These tables are built using different pseudo-random permutations on the n data items outsourced by the client and $\sqrt{n}$ dummy values. Finally, we employ a workspace W of size $n + 2\sqrt{n}$ for constructing incrementally the new table, T_{next}, while the current table, T_{cur}, and the two buffers, B_{cur} and B_{prev}, are being used to process requests. (See Figure 6.1 for illustration.)

Initialization We split a sequence of requests into *epochs*, each consisting of $\sqrt{n}$ requests. Initially, buffers B_{cur} and B_{prev} are empty and each of the tables T_{cur} and T_{next} contains the n items and $\sqrt{n}$ dummy items permuted according to a pseudo-random permutation, where T_{cur} uses permutation π_0 and T_{next} uses permutation π_1.

6.1.2 Access Simulation

Processing an Epoch Buffer B_{cur} caches the $\sqrt{n}$ items being requested in the current epoch while buffer B_{prev} caches the $\sqrt{n}$ items that were requested in the previous epoch. Thus, B_{prev} is empty during the first epoch. Also, table T_{cur} is used for processing requests and workspace W is used to build incrementally a new table, based on a new pseudo-random permutation.

Transitioning to the Next Epoch At the end of an epoch, the new table is copied from W to T_{next}. Next, table T_{cur} and buffer B_{cur} are copied to W. Finally, table T_{next} is copied to T_{cur} to accommodate the requests from the next epoch. Also, we overwrite B_{prev} with items from B_{cur} and we empty B_{cur}.

Incremental Table Construction The construction of the new table, T_{next} in workspace W takes as input T_{cur} and B_{cur} from the previous epoch. We say that the instance of a data item in T_{cur} is stale if there is an instance of the same data item in B_{cur}. Using an algorithm from [21], we obliviously filter out the stale instances of the data items and we construct a table for the set consisting of the n data items and $\sqrt{n}$ dummy items, storing them according to newly generated pseudo-random permutation. Since this algorithm performs $O(n \log^2 n)$ accesses to the data repository, we deamortize it by splitting its sequence of accesses to workspace W into $\sqrt{n}$ batches of $c\sqrt{n} \log^2 n$ accesses each, for some constant $c > 0$. The construction of table T_{next} starts at the beginning of the epoch and a batch of accesses is executed after processing each request. Hence, the new table T_{next} is ready by the end of the epoch.

Algorithm 6 Oblivious RAM simulation with our deamortized version of the square-root approach (Section 6.1).

```
Generate pseudo-random permutation function $\pi_{\text{cur}}$
Initialize table $T_{\text{cur}}$ by storing the $n$ data items and $\sqrt{n}$ dummy items according to permutation $\pi_{\text{cur}}$
Initialize $W$ with $n$ data items and $\sqrt{n}$ dummy items
while true do {process the requests in an epoch}
   Generate pseudo-random permutation function $\pi_{\text{next}}$
   request_count ← 1
   while true do {process request for data item $x$}
      found ← $false$
      Scan all the locations in buffers $B_{\text{cur}}$ and $B_{\text{prev}}$. During the scan, if data item $x$ is found, set found ← true.
      if found then
         Access location $\pi_{\text{cur}}(n + \text{request\_count})$ in $T_{\text{cur}}$ {containing a dummy item}
      else
         Access location $\pi_{\text{cur}}(x)$ in $T_{\text{cur}}$ {containing data item $x$}
      end if
      Rewrite $B_{\text{cur}}$, adding or replacing data item $x$
      Execute the next batch of $c\sqrt{n}\log^2 n$ accesses to workspace $W$ to construct table $T_{\text{next}}$ using permutation $\pi_{\text{next}}$
      request_count ← request_count + 1
      if request_count > $\sqrt{n}$ then
         break {end of the epoch}
      end if
   end while
   Copy the new table from $W$ to $T_{\text{next}}$
   Copy $B_{\text{cur}}$ to $B_{\text{prev}}$
   Copy $T_{\text{cur}}$ and $B_{\text{cur}}$ to $W$
   Empty $B_{\text{cur}}$
   Copy $T_{\text{next}}$ to $T_{\text{cur}}$
   $\pi_{\text{cur}} \leftarrow \pi_{\text{next}}$
end while
```

Our oblivious RAM simulation algorithm based on the square-root approach is outlined in Algorithm 6 and its properties are summarized in Theorem 23.

Theorem 23. *Our oblivious RAM simulation method based on the square-root approach has $O(\sqrt{n}\log^2 n)$ worst-case access overhead per request, $O(\sqrt{n})$ space overhead at the data repository, and $O(1)$ client memory, where n is the number of data items.*

Proof. The worst-case access overhead of each request is $O(\sqrt{n}\log^2 n)$ since we scan two buffers of size $\sqrt{n}$, access one table entry, and execute $O(\sqrt{n}\log^2 n)$ accesses to perform one batch of the table rebuild. Also, $O(n)$ additional space is used at the server.

We now consider the obliviousness of our method. During each epoch, unique items are accessed in table T_{cur}. Namely, if the requested data item is not found in the buffer, we access it in T_{cur}, else we access a new dummy item in T_{cur}. Moreover, in the beginning of each epoch T_{cur} is initialized with a new permutation over n items and $\sqrt{n}$ dummy values.

The method is correct since the user is always returned the most up-to-date instance of the requested item: if the requested data item was last requested in the current epoch then it is found in B_{cur}, else if it was last requested in the previous epoch, it is found in B_{prev}, else T_{cur} has the latest instance. $\square$

Read/Write Data Repository In Algorithm 6 we assumed that the data repository allows us to manage outsourced memory using a *copy* operator. However, we can achieve the same worst case overhead of $O(\sqrt{n}\log^2 n)$ without this operator. We provide some intuition behind this approach. If the data repository supports only read and write operations one can alternate blocks of memory used for rebuilding and for handling requests between the epochs, e.g. during even numbered epochs B_{cur} is used to cache current requests while during odd epochs it serves as a buffer of requests from the previous epoch.

6.2 The $\log n$-Hierarchical Scheme

We now describe the deamortization of our oblivious RAM simulation method presented in Chapter 4, which is based on a hierarchical memory layout at the server and has $O(\log n)$ amortized access overhead. The intuition behind the deamortized version of this method is similar to that for the square-root solution: we incrementally rebuild tables while handling requests using previous versions of tables and buffers. Requests are handled using two sets of buffers: one for items requested in the current epoch and the other for items requested in the previous epoch. However, recall that the construction of the hierarchical solution has $O(\log n)$ buffers implemented either as hash tables with buckets or cuckoo hash tables. This complicates our task since now we need to have a copy of each hash table and cuckoo hash table. Also due to the dynamic arrangement of the buffers, one buffer spills into the next one and so on. We eventually need to construct $O(\log\log n)$ hash tables with buckets and $O(\log n)$ cuckoo hash tables during each epoch.

6.2.1 Setup

Memory Layout Our memory layout at the data repository is schematically illustrated in Figure 6.2. Extending the oblivious RAM data structure of the scheme in Chapter 4), we employ two caches, C_{cur} and C_{prev}, of size $q = O(\log n)$, one stash, S, of size $O(\log n)$, $2l$ hash tables $H_1, H'_1, H_2, H'_2, \ldots, H_{l-1}, H'_{l-1}, H_l, H'_l$, and $2(L-l)$ cuckoo hash tables $T_{l+1}, T'_{l+1}, T_{l+2}, T'_{l+2}, \ldots, T_L, T'_L$. Recall from Section 4 that $l = O(\log\log n)$ and $L = O(\log n)$ and every H_{i+1} can fit twice more elements than H_i, for $i \in \{1, \ldots, l-1\}$, T_{l+1} twice more than H_l, and T_{j+1} twice more than

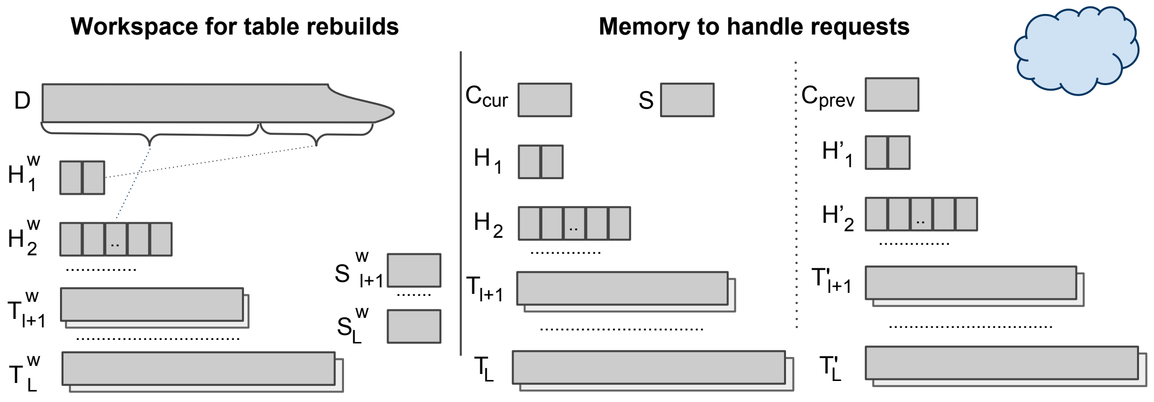


Figure 6.2: Memory layout of the data repository during oblivious RAM simulation of our deamortized version of the $\log n$-hierarchical solution in Section 6.2. (See Figure 4.1 for comparison with the original solution.)

T_j, for $j \in \{l+1, \ldots, L-1\}$. Finally T_L can fit all n elements. The expansion factor for H'_i and T'_j is the same.

We also keep a workspace W for rebuilding each level. The workspace stores the last $2^L q$ requested items in a list, D. In addition, it contains L work areas for rebuilding the hash tables. We proceed with the explanation for the levels that contain cuckoo hash tables, while similar ideas apply for hash tables with buckets as well. The i-th work area consists of storage space T_i^W of size $O(2^i q)$ for a cuckoo hash table at level i, and of overflow space S_i^W of size $O(\log n)$ for the corresponding stash.

Initialization We build T_L as a cuckoo hash table for the n data items and put into stash S any items that did not fit. Both caches C_{cur} and C_{prev} and other tables H_i, H'_i (for $1 \le i \le l$), T_i and T'_i (for $l \le i < L$) are empty.

6.2.2 Access Simulation

Processing an Epoch In this section an epoch is defined as a sequence of q requests. During an epoch, cache C_{cur} stores data items last requested in the current epoch and cache C_{prev} stores data items last requested in the previous epoch. Thus, these caches play roles similar to those of buffers B_{cur} and B_{prev} in Section 6.1. Each request is processed by scanning caches C_{cur} and C_{prev}, scanning stash S, and accessing buckets of H_1, H'_1, H_2, H'_2, …, H_l, H'_l, and locations in tables T_{l+1}, T'_{l+1}, …, T_{L-1}, T'_{L-1}, T_L. In addition, a batch of accesses is made to workspace W toward rebuilding its cuckoo tables. The incremental rebuilding process guarantees the completion of a cuckoo table in T_i^W and its stash in S_i^W after 2^{i-1} epochs.

Incremental Construction of L Cuckoo Hash Tables Recall that a cuckoo hash table T_i of size $2^i q$ can be constructed obliviously using $2^i q$ accesses to the data repository and $O(n^\nu)$ private memory [24]. To help us explain the concurrent oblivious rebuild of L cuckoo hash tables, we introduce a data structure I that stores the set of indices of the cuckoo tables that need to be rebuilt in workspace W. Note that *Queue* depends only on the number of requests made so far hence it can be computed in constant time. I starts empty. After every 2^{i-1} epochs index i is added to I. When an index i is added to I a sequence of $2^i bq$ accesses is required for a rebuild of T_i^W, for some constant $b > 0$. After each request, the client executes $2b$ accesses for each index in I so that the construction of table T_i^W is completed in 2^{i-1} epochs. Observe that after the first 2^{i-1}, epochs index i is always present in I. Moreover, after 2^{L-1} epochs indices $1, 2, \ldots, L$ are present in I and I does not change from then on. This also means that eventually all L tables are being rebuilt during an epoch. (See Figure 6.3 for an illustration of the rebuilding process.) To accommodate L concurrent rebuilds, we increase the requirement on the size of

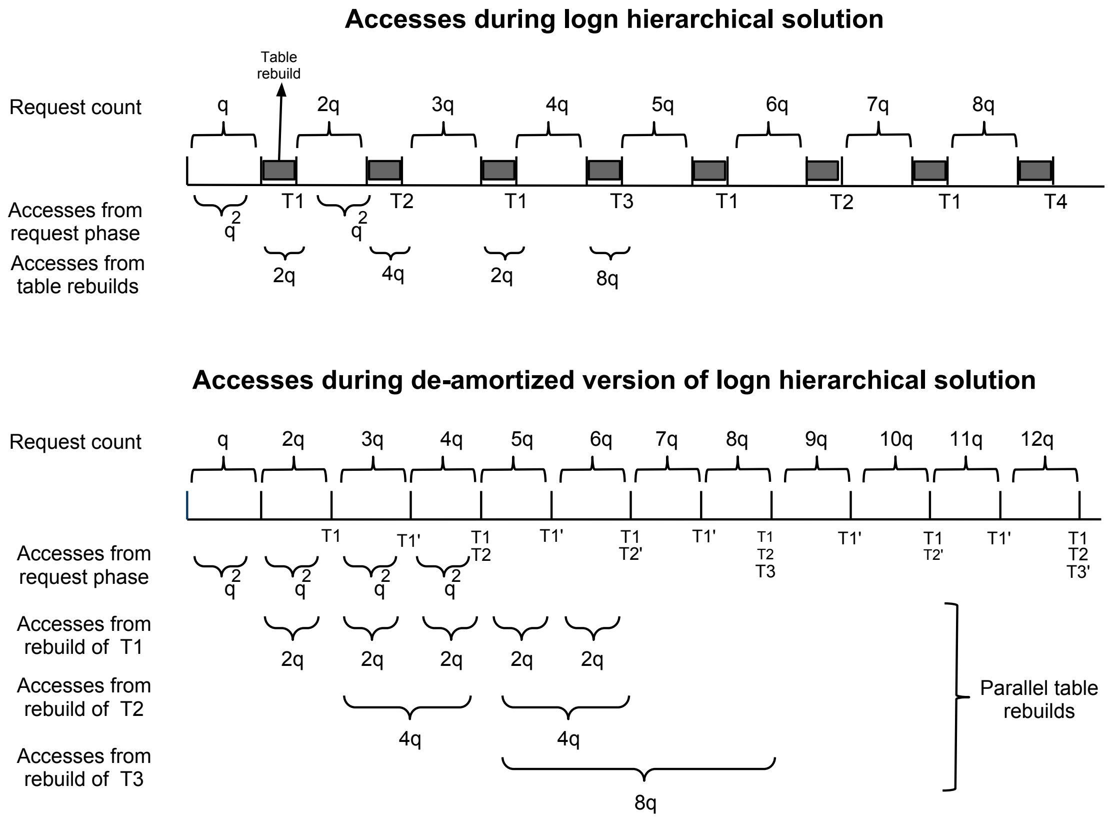

Accesses during logn hierarchical solution
Table rebuild
Request count
q
2q
3q
4q
5q
6q
7q
8q
T1
T2
T1
T3
T1
T2
T1
T4
Accesses from request phase
q^2
q^2
Accesses from table rebuilds
2q
4q
2q
8q
Accesses during de-amortized version of logn hierarchical solution
Request count
q
2q
3q
4q
5q
6q
7q
8q
9q
10q
11q
12q
T1
T1'
T1 T2
T1'
T1 T2'
T1'
T1 T2 T3
T1'
T1 T2'
T1'
T1 T2 T3'
Accesses from request phase
q^2
q^2
q^2
q^2
Accesses from rebuild of T1
2q
2q
2q
2q
2q
Accesses from rebuild of T2
4q
4q
Accesses from rebuild of T3
8q
Parallel table rebuilds

Figure 6.3: Sequence of accesses during oblivious RAM simulation: (a) original version of the $\log n$ hierarchical solution (Section 4); (b) our deamortized version (Section 6.2). The size of the cache is denoted by q, which is $O(\log n)$.

client's private memory from $O(n^{\nu})$ in Chapter 4 to $O(n^{\tau})$, for some fixed constants $\tau > \nu$ and $\nu > 0$.

Transitioning to the Next Epoch We append items in C_{cur} to D, the list in workspace W that keeps track of the $2^L q$ previously requested items. We then copy C_{cur} to C_{prev} and empty it. Hence C_{cur} can be used to cache requests during the next epoch. We then check which tables are finished, i.e. T_i^W is finished if the current number of epochs is a multiple of 2^{i-1} since T_i^W takes 2^{i-1} epochs for a rebuild. Each such table T_i^W is then copied to either T_i or T_i'. If T_i and T_i' are both empty or both full T_i^W is copied to T_i, stash S_i^W is merged with S and T_i' is cleared. If only T_i is full T_i^W is copied to T_i', stash S_i^W is merged with S. If T_L, the table from the last level, is finished we clear first $2^L q$ items from D since all these items are now in T_L and no table from earlier levels requires them for a rebuild.

Stash Size In Section 4.3 we show that a single stash of size $O(\log n)$ is enough to avoid overflows in cuckoo hash tables T_{l+1}, …, T_L where T_i contains $2^i q$ items. In this construction, we use a single stash S for two collections of cuckoo hash tables. A single stash ensures that if item x happened to not fit into two tables T_i and T'_j then only the most recent copy is present in S. One can view stash S as a joint stash between tables $T_{l+1}, \ldots, T_L$ and $T'_{l+1}, \ldots, T'_{L-1}$. Suppose a stash of size $s \log n$ is used in the $\log n$-hierarchical construction, where $s > 1$ is a constant. Then we set our stash S to be of size $2s \log n$, where the first $s \log n$ locations are used for tables $T_{l+1}, \ldots, T_L$ and the remaining $s \log n$ locations for $T'_{l+1}, \ldots, T'_{L-1}$, with the additional constraint that only unique items can be present in S. The latter constraint is enforced when we merge stash S_i^W of a new table T_i^W with S.

Our oblivious RAM simulation algorithm based on the $\log n$ hierarchical approach is outlined in Algorithm 7 and its properties are summarized in Theorem 24.

Theorem 24. *Our oblivious RAM simulation method based on the* $\log n$ *hierarchical approach has* $O(\log n)$ *worst-case access overhead per request,* $O(n)$ *space overhead at the data repository, and* $O(n^\tau)$ *client memory, where* n *is the number of data items and* τ *is any fixed positive constant.*

Proof. We first show that handling of each request using the above protocol takes $O(\log n)$ accesses. Retrieving a data item takes $O(\log n)$ accesses since three blocks of size $O(\log n)$ are scanned and two accesses are made to $2L$ tables, where L is $O(\log n)$. The batch of accesses for table rebuilding made after each request consists of $2b$ accesses for every table in I, where I has at most L indices and b is a constant.

The method clearly requires only $O(n)$ space on the data repository. For every rebuild, we use the method of [24] which requires $O(n^{\nu})$, $\nu > 0$, of client private memory. Since our method concurrently makes $O(\log n)$ rebuilds $O(n^{\tau})$ of private memory is required for $\tau > \nu$.

We now consider the obliviousness of the method. Table rebuilds remain oblivious since they follow a predetermined schedule that depends on n and request_count and are performed in the same way as in the original ORAM construction in Section 4.

It remains to show that each request remains oblivious. Accesses to the caches, C_{cur} and C_{prev}, and the stash, S, are oblivious since their memory locations are scanned (read and rewritten entirely) for each request. Since accesses to table T_i depend on whether an item is found in T_i' we first show the obliviousness of access sequence to T_i'. Observe that when T_i' is substituted with a new cuckoo hash table T_i^W cache C_{cur} and all tables $T_{j<i}'$ are empty. Since each table on level i can store up to $2^{i-1}q$ items before it is emptied there is space to remember the following number of requests:

$$q + \sum_{j=1}^{i-1} 2^{j-1}q = 2^{i-1}q.$$

If an item is not found in previous levels, it is accessed according to pseudo-random hash functions $h_1^{i'}$ and $h_2^{i'}$. Otherwise, T_i' is accessed in random locations. T_i' is cleared as soon as next table for this level, i.e. next T_i^W, is ready. This happens $2^{i-1}q$ requests after T_i' was last substituted with a new table. Hence, T_i' is never accessed more than once for the same item.

An access to table T_i follows an access to T_i' and is random if an item is found in earlier tables or in T_i'. Note that similarly to T_i' cache C_{cur} and all tables $T_{j<i}'$

are empty when T_i is ready. Moreover T_i' is empty as well. Hence, there is space to remember the following number of requests:

$$q + \sum_{j=1}^{i-1} 2^{j-1}q + 2^{i-1}q = 2^i q.$$

However, T_i is replaced with a new table every $2^i q$ requests. Hence, no location is accessed more than once in table T_i as well.

To prove the correctness of the method, we observe that the most current copies of the data items are present in the caches or smaller tables. Moreover, table T_i' contains more recent requests than T_i and stash S contains any items that did not fit in their corresponding tables. When newly constructed tables are moved from W to the memory for handling requests, we merge the stash of larger tables with S first. In this case, if the same item did not fit into more than one table, only the most recent copy is in S. Since we first scan the caches, the stash and start accessing tables from smaller levels, with T_i' before T_i, our method returns to the user the latest instance of the requested item. □

Read/Write Data Repository Similarly to the deamortized version of the square-root solution from Section 6.1, we can relax the assumption on the interface that data repository provides us. If read and write are the only supported operations, we can alternate the blocks of memory used for rebuilds and for handling the requests depending on the epoch count.

6.3 The Oblivious Storage

We now consider how deamortization techniques can be applied to the constant overhead solutions proposed in Section 5. We first consider the square-root based construction in Section 5.4 and then the construction with smaller message size in Section 5.5.

Square-Root OS We recall that the main difference between the square-root solution considered in the beginning of this Chapter and that of Section 5.4 lies in the message size, $O(1)$ vs. $O(\sqrt{n}\log n)$, and the rebuild method, oblivious sorting vs. oblivious shuffle. We proceed by having the same Setup phase as in Section 6.1.1 where we split the server memory in B_{cur}, B_{prev}, T_{cur}, T_{next} and W. The size of W is increased to $O(n\log n)$ since this is the temporary size that is required by the Melbourne Shuffle.

The access phase proceeds as follows. Instead of scanning B_{cur} and B_{prev} the client retrieves the buffers in a single I/Os since their size fits in a single message and into private memory, which is $O(\sqrt{n}\log n)$. The access to T_{cur} remains the same and rewrite of B_{cur} is done in a single I/O. Once the access part of the request is done we can proceed with $O(1)$ steps of the Melbourne Shuffle.

Cth-Root OS The deamortization of the construction with $O(\sqrt[c]{n}\log n)$ messages follows from extending the deamortization of the $\log n$-hierarchical solution in Section 6.2, where every level is doubled to store the copy of the hash table with buckets from the previous access phase. The rebuild accesses are made according to several iterations of the Melbourne Shuffle with small messages.

Algorithm 7 Oblivious RAM simulation with our deamortized version of the $\log n$ hierarchical approach (Section 6.2). For simplicity we remove references to the first l hash tables with buckets from the pseudo-code.

```
Initialize $T_L$ and $S$ with a cuckoo hash table with a stash using $n$ data items.
$I \leftarrow \{\}$, request_count $\leftarrow 0$
**while** true **do**
  **while** true **do** {on request $x$}
    $found \leftarrow false$
    Scan all the locations in caches $C_{\text{cur}}$ and $C_{\text{prev}}$ and stash $S$. During the scan,
    if data item $x$ is found, found $\leftarrow$ true.
    **for** each level $i$, $1 \leq i \leq L$ **do**
      **if** $i \neq L$ and $T'_i$ is not empty
        if $found$ is $true$ access random locations in $T'_i$.
        else access locations $h_1^{i'}(x)$ and $h_2^{i'}(x)$ in $T'_i$.
        if $x$ is found, $found \leftarrow true$.
      **if** $T_i$ is not empty
        if $found$ is $true$ access random locations in $T_i$.
        else access locations $h_1^{i}(x)$ and $h_2^{i}(x)$ in $T_i$.
        if $x$ is found, $found \leftarrow true$.
    **end for**
    Rewrite $C_{\text{cur}}$, adding or replacing data item $x$.
    if $x$ is found in stash $S$ remove $x$ from $S$. Rewrite $S$.
    **for** $i \in I$
      Make next $2b$ accesses towards a rebuild of table $T_i^W$
    request_count $\leftarrow$ request_count $+ 1$
    **if** request_count $\bmod q = 0$ **then**
      {end of the epoch}
      Copy $C_{\text{cur}}$ to $C_{\text{prev}}$ and append it to $D$.
      Empty $C_{\text{cur}}$.
      **for** $i \in sorted\_decr\_order(I)$
        **if** request_count $\bmod 2^{i-1}q = 0$
          **if** $i = L$
            copy $T_i^W$ to $T_L$.
          **else if** $T_i$ and $T'_i$ are both full or both empty
            Empty $T'_i$ and copy $T_i^W$ to $T_i$.
          **else**
            copy $T_i^W$ to $T'_i$.
          Merge $S_i^W$ and $S$.
      **for** each level $i$, $1 \leq i \leq L$
        **if** request_count $\bmod 2^{i-1}q = 0$
          Copy last $2^{i-1}q$ items from $D$ to $T_i^W$
        **if** $i \notin I$
          $I \leftarrow I \cup \{i\}$.
        **if** $i = L$
          empty $D$.
    **end if**
  **end while**
**end while**
```

Chapter Seven

Hiding Access Patterns of Graph Drawing Algorithms

So far in this thesis we have looked at general access simulations that can be applied to an arbitrary algorithm and simulate its request sequence obliviously. But as we saw these solutions involve fairly complicated simulation techniques for generic algorithms that increase the running time of the client by a polylogarithmic factor when the client has a small workspace. In this chapter, we instead study oblivious algorithms that protect the pattern of a specific application. In particular, we design algorithms to protect the access patterns that result from the application of graph drawing to a cloud-computing context where data is stored externally and processed using a small local storage. We show that a number of classic graph drawing algorithms can be efficiently implemented in such a framework where the client can maintain privacy while constructing a drawing of her graph.

7.1 Overview

We are interested in allowing a client to access her graph and perform computations on it in a *privacy-preserving* way. For example, an administrator for a fast-growing company may be revising (and visualizing) the organizational chart for the leadership of her company, and leaking this chart to the press or a rival could negatively impact the company. As in previous constructions we assume that the client encrypts her graph before outsourcing it. She is also interested in hiding the pattern in which she accesses the graph. For example, accessing the memory associated with a certain department while preparing a new organizational chart leaks the fact that that department is being reorganized.

Our aim is to develop simple privacy-preserving algorithms for some classic graph drawing problems that fully obfuscate the access pattern from the data server. To enable data-oblivious algorithms for graph drawing problems, we introduce *compressed-scanning*, an algorithmic design framework based on a series of scans. Our method is related to the massive, unordered, distributed (MUD) model [16] for efficient computation in the map-reduce framework. We assume that the server holds a set of n data items and the client has a small private workspace of size $O(\log n)$. The data items at the server are encrypted with a semantically secure encryption (Section 3.3) so that it is hard for the server to determine whether two items are equal.

An algorithm for the compressed-scanning model consists of a sequence of rounds, where in each round the entire data set is scanned by the client. During the scan, each item is processed exactly once by the client: first the client downloads the item from the server into workspace; next, the client performs some internal-memory computation on the item and the content of the workspace; finally the item is written out to an output stream at the server. When a round is completed, the output stream is either confirmed as the algorithm's output or it is used as the input data set for the next round. The efficiency of such an algorithm is measured, therefore, by the number of rounds needed and the size of the local workspace that is required. Ideally, the number of rounds should be $O(1)$ and the workspace should be sublinear. As shown in Section 7.2.2, an algorithm designed in the compressed-scanning framework can be implemented in a data-oblivious way by randomly shuffling the items in between scans.

Using the compressed-scanning approach, we provide efficient data-oblivious algorithms for a number of classic graph drawing methods [12], including symmetric straight-line drawings and treemap [30] drawings of trees, dominance drawings of planar acyclic digraphs [13], and Δ-drawings of series-parallel graphs [6]. Our methods result in privacy-preserving graph drawing algorithms with a smaller overhead than could be achieved by applying general-purpose privacy-preserving techniques we discussed so far.

7.2 Compressed-Scanning Model

In this section, we formally define the *compressed-scanning* model for designing client-server algorithms that can be efficiently implemented using a small workspace, W, at the client. We assume that the server holds an array, S, of n data elements.

7.2.1 Model

An algorithm for our model consists of a sequence of t rounds. A round involves accessing each of the elements of S exactly once in a read-compute-write operation. This operation consists of reading an element from the server into private workspace, using the element in some computation, and writing a new element to an output stream, O, at the server. When a round completes, either the output stream O and/or a set of values in W are confirmed as the output of the algorithm, or we assign $S = O$ and start the next round. Hence, the running time of an algorithm in our model is $O(tn)$.

This size of the workspace, W, is a parameter of our model, and is intended to be small (e.g., constant or $O(\log n)$). The name of our model is derived from the fact that each round scans the set S and computations are performed using a small, or "compressed", amount of workspace. Simple examples of algorithms that fit our model include the trivial methods for summing n integers in an array or traversing a linked list from beginning to end, which can be done with a constant-size workspace, or any algorithm in the standard data streaming model, which would have W being equal to the workspace for that algorithm.

7.2.2 Privacy Protection

Suppose we are given a compressed-scanning algorithm, A, which runs in t rounds using a workspace, W, and a data set, S, of size n. We can implement A in a privacy-preserving way as follows. The input stream, S, is stored encrypted at the server and whenever we write elements to the output stream, O, we also encrypt them. Hence, the server will not be able to distinguish whether two data elements are equal or whether the output element of a read-compute-write operation is equal to the input element.

The next step in ensuring privacy is hiding the access pattern from the server. In other words, the accesses to S should be data independent in each round and one should not be able to correlate accesses between the rounds. Each round in our model consists of scanning S one element at a time, performing local computations using the value of this element, and possibly modifying it and writing it back. Even if we have nothing to output, we can always write a dummy element, for the sake of being oblivious. However, a single scan is not enough to perform complex computations over data. The computation in the next round usually relies on computations from previous rounds and may require rearrangement of the data to allow a sequential access of that round. This shuffle of the data can be carried out by sorting over one of element's fields.

We employ an oblivious sorting algorithm for the purpose of hiding the correlation of accesses between the rounds. As mentioned in Section 3.6, several oblivious sorting techniques have been developed. Each oblivious sorting algorithm B offers a tradeoff between the time it takes the client to sort n items, $\mathsf{sort}_B(n)$, and the size of the client's private workspace, $\mathsf{workspace}_B(n)$. In oblivious Batcher's sorting network, either $\mathsf{sort}_B(n)$ is $O(n \log^2 n)$ and $\mathsf{workspace}_B(n)$ is constant or $\mathsf{sort}_B(n)$ is $O(n)$ and $\mathsf{workspace}_B(n)$ is $O(\sqrt{n})$ [24]. In oblivious randomized shell short [23], which succeeds with high probability, we have that $\mathsf{sort}_B(n)$ is $O(n \log n)$ and $\mathsf{workspace}_B(n)$ is constant. We can use one of the above methods depending on the tradeoff we are willing to take and from now on, we refer to the oblivious algorithm as a black box algorithm, B.

In conclusion, our simulation of algorithm A consists of t scans of S and a call to an oblivious sort procedure B between the rounds. Each round requires $O(\mathsf{sort}_B(n))$ time while fully hiding the pattern of access to the items in S. Thus, the simulation of A takes time $O(t\ \mathsf{sort}_B(n))$ and uses workspace of size proportional to that of A plus the space required between the rounds for sorting, $\mathsf{workspace}_B(n)$.

Theorem 25. *Let A be an algorithm in the compressed-scanning model for an input of size n that uses a workspace of size $\mathsf{workspace}_A(n)$. Algorithm A can be simulated by a data-oblivious algorithm if the number of rounds and the number of elements written to the output stream at each round depend only on n. Also, the simulation uses a workspace of size $O(\mathsf{workspace}_A(n) + \mathsf{workspace}_B(n))$ and runs in time $O(t\ \mathsf{sort}_B(n))$, where t is the number of rounds of A and B is an oblivious sorting algorithm.*

Proof. (*Sketch*) Each round is simulated by reading elements from S, writing elements to O, and reshuffling the next input set. Accesses to locations in S are made in a sequential order. This ensures that accesses to S in a single round are data-oblivious. Write accesses to O are also data-oblivious, since they happen on every access to S. After every round, the input sequence is reshuffled (data-obliviously); hence, one cannot correlate accesses between rounds as well. Thus, accesses to S and O depend only on size of S while the number of rounds is fixed by the algorithm regardless of S. □

In the next section we describe graph drawing algorithms that fit the compressed-scanning model and, hence, can be implemented in a data-oblivious manner.These algorithms guarantee that their access patterns do not reveal the combinatorial structure of the graphs that are given as inputs (e.g., number of outgoing or incoming edges for a particular node) and run in a constant number of rounds using W of logarithmic size.

7.3 Graph Drawing Algorithms

Most existing graph drawing algorithms are designed without privacy concerns in mind; hence, if they are run in a cloud-computing environment, they can reveal potentially sensitive information from their access patterns. For example, a recursive binary-tree drawing algorithm implemented in the standard way can reveal the depth of the tree from the access patterns used for the recursion stack, even if all the nodes in the tree are encrypted. In this section, we present several graph drawing algorithms modified to fit the compressed-scanning model. In order to build a graph drawing algorithm that fits this model, we modify the representation of the graph so that we never access the same location more than once in the same round. For example, consider a tree represented with a set of nodes and pointers from each node to its children and a parent. Traversing the tree in this case involves accessing an internal node several times depending on its degree, which reveals information about the tree.

7.3.1 Euler Tours in the Compressed-Scanning Model

Traversing a tree in the compressed-scanning model requires that we access each memory location exactly once; hence, we need to reorganize how we normally perform data accesses, since, for example, we cannot access a parent again when coming from its left child after we have already visited it. Given small private storage, W, we cannot store previously accessed nodes. Thus, we need a representation of a tree that allows for a traversal where elements are accessed only once. For this purpose, we construct an Euler tour over a tree that is based on duplicating edges and defines a left to right traversal of a tree. Each copy of an edge contains a pointer to a copy of the next edge in the tour so we can go to the next edge without using recursion and visiting each edge of the tour only once.

For an ordered tree, $T = (V, E)$, we store an Euler tour as a set of items, C, where $|C| = 2|E|$. Each item represents an edge of the tour and stores information related to the tree, e.g., parent, child node names, and the order of the child among all its siblings. Additionally, it stores information related to the actual cycle of the Euler tour: (a) tag: a unique tag associated with this item, $0 \leq \text{tag} < 2|E|$. This is used to locate and sort the items. (b) direction: up or down. This indicates which direction in the tree we are following. (c) next: tag of the next edge in the cycle.

We assume that tag $= 0$ for the leftmost edge of the root of T. Suppose we shuffle the items in C using the tag field. Then a traversal of C is a simple scan of the memory and is data-oblivious revealing only the number of edges and nodes in the tree.

7.3.2 Computation over Euler Tour Representations

Many graph drawing algorithms collect information from a tree representation of the graph to determine the layout. Such information could be the height, width, or subtree size of each node of the tree. We now show how one can use an Euler tour representation of a rooted tree to compute for each node of the tree, the size (number of nodes) of its subtree in a data-oblivious manner.

For this computation, we add a new field $\mathsf{subsize}$ for every edge in the Euler tour C. The algorithm maintains in local memory, W, a variable, $\mathsf{total\_subsize}$, initially set to 0. Edges in C are traversed as described in the previous section. However, every time we now read an edge, i, we update $i.\mathsf{subsize}$ with the value stored at $\mathsf{total\_subsize}$ and write it back. When we are going up, i.e., $i.\mathsf{direction} = \mathsf{up}$, $\mathsf{total\_subsize}$ is incremented by 1. Once the traversal finishes, we observe that for every two items, i and i', that represent a traversal of the same edge, i.e., $i.\mathsf{parent} = i'.\mathsf{parent}$, $i.\mathsf{child} = i'.\mathsf{child}$, $i.\mathsf{direction} = \mathit{down}$ and $i'.\mathsf{direction} = \mathit{up}$, the value $(i'.\mathsf{subsize} - i.\mathsf{subsize})$ is the size of the subtree rooted at $i.\mathsf{child}$ and the final value of $\mathsf{total\_subsize}$ is $\mathsf{subsize}$ of the root. However, we need to associate nodes of the tree T with these values in the compressed-scanning model as well. For this purpose, we obliviously sort the values in C using the fields, parent and child, to bring items that correspond to the same edge next to each other. We then simply scan the resulting sorted list and after reading a pair of items, i and i', output a pair $(i.\mathsf{child}, i'.\mathsf{subsize} - i.\mathsf{subsize})$. (See Figure 7.1.)

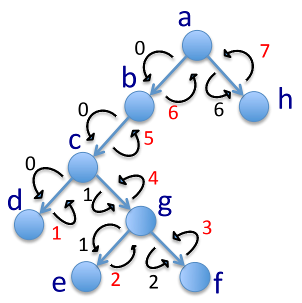


Figure 7.1: Computing the size of the subgraph via Euler tour. During the tour a locally maintained variable total_subsize is incremented when the tour goes up the tree (red numbers in the figure) and is assigned to currently visited edge. The size of the node's subgraph is the counter at the edge going up from this node minus the counter of the duplicate of this edge. For example, the size for g is (4-1) = 3.

The above computation consists of two rounds: the first round reads one item of C at a time, modifies it and writes it back. The second round starts after the sorting is complete, where items are read one at a time and a new item is written to the output after every two reads. We can compute the depth of each node using a similar technique.

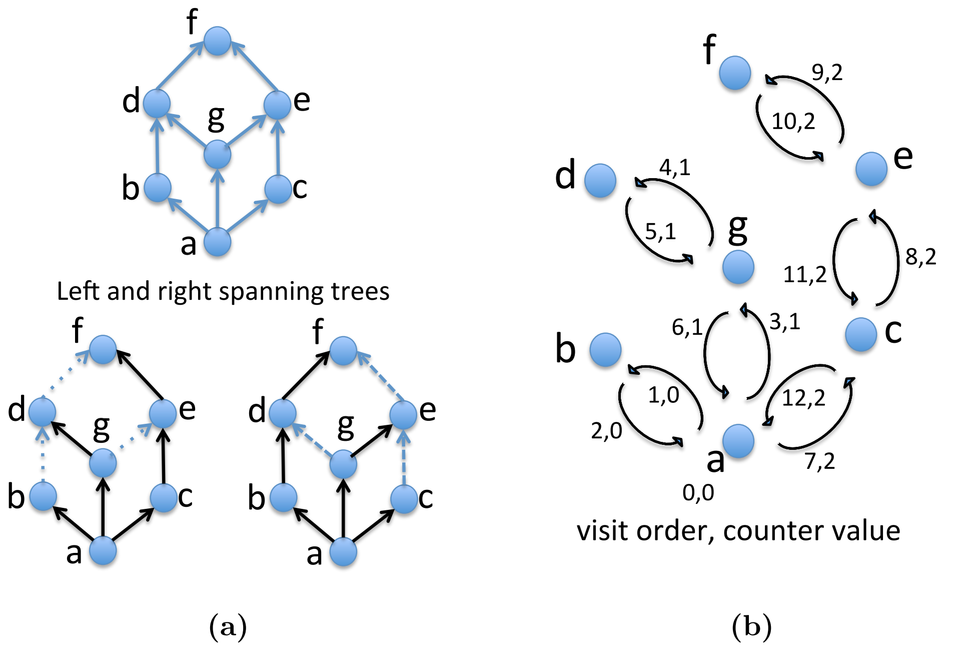
7.3.3 Drawing of Planar Acyclic Digraphs

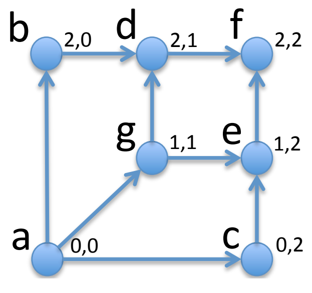
We adopt an algorithm for dominance drawings of planar acyclic st-digraphs from [13], which is simple and elegant but is not data-oblivious. To find the x-coordinate of each node, one builds a spanning tree based on leftmost incoming edges of the nodes and then traverses this tree from left to right, numbering each node in this order. The resulting numbering of each node is its x-coordinate. The algorithm to determine the y-coordinates uses the rightmost spanning tree.

Input: We assume that the graph, G, is given as a set of edges, E, where $e \in E$ is an edge directed from node a to b storing indegree, the number of incoming edges to b, and child_num, the order of a among all incoming edges to b; the leftmost edge has order 0.

Data-oblivious algorithm: Following the original algorithm, we show how one can construct a spanning tree and number the nodes to get the final drawing. Our first task is to augment each edge with information about a spanning tree of G. We augment e with additional fields, left_spanning and right_spanning, which are set to true or false depending on which spanning tree e belongs to. In the compressed-scanning model, one simply accesses e, sets e.left_spanning to true if e.indegree equals e.child_num or e.right_spanning to true if e.child_num is 1, and writes e back.

Given annotated edges, we construct an Euler tour over each spanning tree. Note that given that the number of nodes in G is revealed, we do not need to hide the number of edges in either of the spanning trees. For ease of explanation, we say that we traverse an edge down when we follow an edge of the spanning tree in its direction in G. The left spanning tree is traversed starting with the leftmost outgoing edge of the root, and rightmost outgoing edge for the right tree. We are now ready to make a tour traversal and assign coordinates to the nodes. We adopt a compressed version of the algorithm that minimizes the area of the drawing and start with traversal of the left tree. In private memory, a counter for x-coordinates is maintained, set to 0. Initially, we output $(\mathsf{source}, 0, x)$. For every edge e that has $\mathsf{direction} = \mathsf{down}$, and $e.\mathsf{indegree} > 1$ or e is the first traversed edge of a, we output $(e.b, \mathsf{counter}, x)$. If e has down direction but is not the first edge of a traversed (in Euler tour this corresponds to remembering the latest visited edge) or is the only incoming edge to b, then we increment the counter by 1 and output $(e.b, \mathsf{counter}, x)$. If e.direction is set to up, then we output $(\mathsf{dummy}, 0, x)$. The algorithm for computing y-coordinates is similar and outputs values with $(e.\mathsf{parent}, \mathsf{counter}, y)$. Note that access pattern of reads and





(c)

Figure 7.2: (a) A planar acyclic st-digraph with its left and right spanning trees. (b) The order of the visit to each edge of Euler tour of the left spanning tree and the counter of x coordinate for child nodes, e.g., edge a-g is visited third and g is assigned x coordinate of 1. (c) The final drawing.

writes is always the same: read an edge of the Euler tour and output a tuple of three values.

The output of the above procedure contains tuples of real and dummy values. We can remove dummy values and bring x, y coordinates of each node together by obliviously sorting tuples by the first field (node name) such that string dummy is always greater than any real node name. The resulting list contains all dummy tuples at the end. Also, each node has its x- and y-coordinates adjacent. See Figure 7.2 for an example.

7.3.4 Treemap Drawings

Treemaps are a representation designed for human visualization of complex tree structures, where arbitrary trees are shown with a 2-d space-filling area. Here, we present how one can draw a treemap using an algorithm from [30] adapted to the compressed-scanning model. The original algorithm takes a rectangle area and splits it vertically into two sections. The area of the first section is enough to fit the first child, child_1, of the root and the rest is enough to fit the rest of its children. The next step is to divide the first section among children of child_1 but this time splitting the area horizontally. The algorithm continues in the same manner for all decedents of child_1. Once finished, it proceeds to splitting the second section between second child of the root, child_2, and the rest of root's children.

Input: A tree, T, where each leaf also contains a value area and the size of a rectangle area, $w \times h$, where T should be drawn. We build an Euler tour, C, from T and add two fields parent_area and child_area to each edge in C.

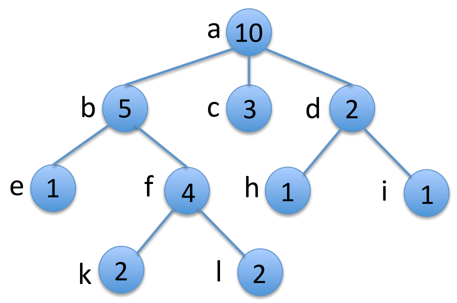
Output: Each node is labeled with (x, y) coordinates of the top-left corner, P, and bottom-right corner, Q, of the rectangle area where the node should be placed in.

Data-oblivious algorithm: We first run a procedure similar to the one for computing subsize, to assign area values to inner nodes of T. The original algorithm labels the nodes with values P and Q via pre-order traversal of T. The algorithm we propose here first goes down the leftmost subtree computing values P, Q and labeling the nodes on the way. In private memory, it maintains only one copy of the last two assigned values of P and Q, prevP and prevQ. It then goes up the tree "undoing" all the computations made to prevP and prevQ. We do it in such a way that when going up and reaching some node, we recover its P and Q values as they were before we visited any of its children or other nodes in its subgraph. This algorithm fits the traversal of Euler tour C of the tree T. When going down the tree, we read each item i of tour C and output P, Q values corresponding to $i.\mathsf{child}$. However, when going up we cannot retrieve earlier written P, Q values, since this will not be data-oblivious and we reveal that we are going up, which consequently reveals the depth of the tree. This is where "undoing" computations when going up on prevP and prevQ helps. This is possible since the information used to compute P and Q is stored twice in C: once for edge with direction set to down and once for up. The pseudocode of the algorithm appears in Algorithm 8. Figure 7.3 shows an execution of the algorithm on a small tree.

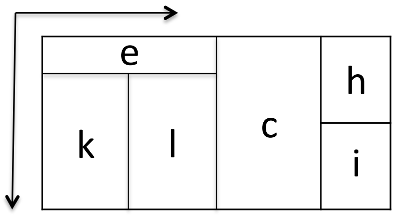
7.3.5 Series-Parallel Graphs

A series-parallel (SP) graph is a directed acyclic graph that can be decomposed recursively into a combination of series-parallel digraphs. The base case of such a graph is a simple directed edge. A series composition consists of two series-parallel graphs G_1 and G_2 where the sink of G_1 is identified with the source of G_2. A parallel composition of two series-parallel graphs G_1 and G_2 is the digraph where source of G_1 is identified with the source of G_2 and similar for their sink nodes. For example, consider the series-parallel digraph shown in Figure 7.4a. The subgraph S' induced by its edges c-d and d-a is a series composition of graphs c-d and d-a. While S' and edge c-a is a parallel composition.

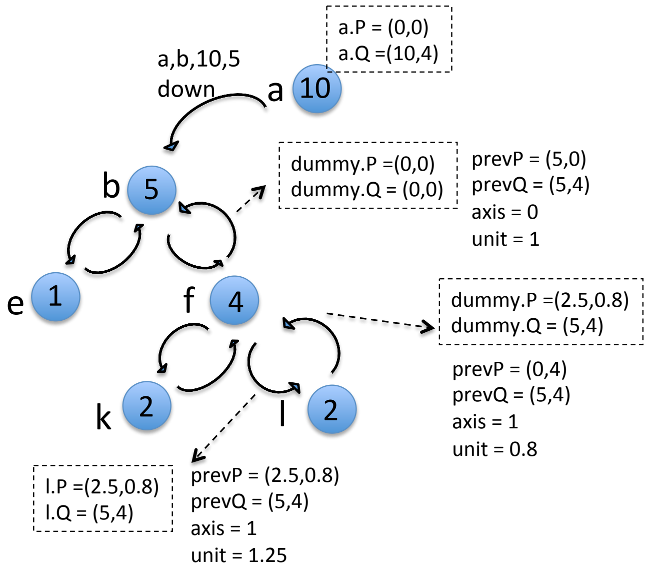
An SP graph G can be represented with a binary tree (SPQ tree) with three types of nodes, S, P and Q. Q nodes are leaves of the tree and correspond to individual edges of G. An internal node is of type P if it is a parallel composition



(a)



(b)



(c)

Figure 7.3: Treemap graph drawing. (a) The original graph. (b) The final drawing. (c) Execution of algorithm in Section 7.3.4 on the graph in (a) on a 10×4 rectangle area. The values in dashed rectangles are written for every edge and are never accessed. Variables prevP, prevQ, axis and unit are kept in memory.

of the children digraphs. If a node corresponds to a series composition it is called S node. Here, we use a right-pushed embedding of G such that a transitive edge in parallel composition is always embedded on the right. (Figure 7.4b shows the SPQ tree of the graph of Figure 7.4a.)

Original Δ-drawing algorithm: We adopt the Δ-drawing algorithm from [6]. This algorithm recursively produces a drawing of G inside a bounding triangle $\triangle(G)$ which is isosceles and right-angled. In the drawing of a series composition, the two bounding triangles, $\triangle(G_1)$ and $\triangle(G_2)$, are placed one on top of another and, hence, produce a bounding triangle big enough to fit them both. For a parallel composition, $\triangle(G_2)$ is placed on the right of $\triangle(G_1)$ and a larger triangle is drawn to fit this parallel composition. The algorithm works by traversing the SPQ tree and identifying the size of the bounding triangles of each node. The length of the hypotenuse, b, is enough to store this information. Each Q node is assigned a triangle with $b = 2$, while for series and parallel nodes b is the sum of b values at the children nodes. When traversing the tree we also compute value b', which makes sure that in a drawing of a parallel graph G the edge that goes from the source of G to G_1, the left subgraph of the composition, does not intersect the drawing of G_2. This value b' for a Q node is simply b, for S node it is $b'(\triangle(G_1))$ and for P node it is $b'(\triangle(G_1)) + b'(\triangle(G_2))$. Note that for a parallel node it is the sum of b' values of both graphs since we want to make sure that if subgraph G is later a part of a parallel composition no node will intersect either G_1 or G_2. If G is a transitive edge then $b'(\triangle(G)) = b(\triangle(G))$. (See Figure 7.4b.)

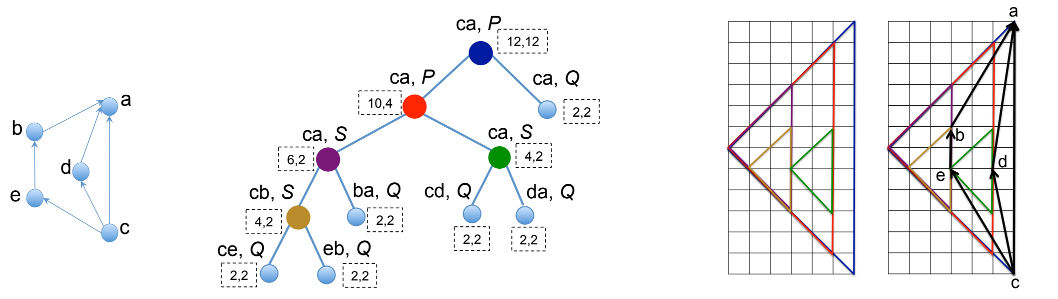
Once b and b' are computed for every node, i.e., every bounding triangle, the algorithm computes the (x, y) value of the bottom node of each triangle. The outer most triangle is positioned at $(0, 0)$. Given coordinates (x, y) of a triangle corresponding to the S node with hypotenuse of size b and children with hypotenuses b_1 and b_2, we place the first triangle at (x, y) and second at $(x, y + b_1)$. Given coordinates (x, y) of a triangle corresponding to a parallel node, we place the first triangle at $(x - 0.5b_2, y + 0.5b_2)$ and second at $(x, y + b'_1)$. Given that we know the coordinates of each triangle, we can now assign coordinates for individual nodes. The source of G is placed at (0,0) and sink is placed at $(0, b(\triangle(G)))$. We then look at each node in G and place it at $(x, y + b(\triangle(G_{\mathsf{node}})))$ where G_{node} is a subgraph and node is its sink. (See Figure 7.4c for an example.)

We are now ready to explain the algorithm in compressed-scanning model.

Input: SPQ tree from a right-pushed embedding of SP digraph G and nodes that are annotated as S, P or Q. We convert this tree into an Euler tour with addition of parent and child node type: $\mathsf{parent\_spq\_type}$ and $\mathsf{child\_spq\_type}$ which are either S, P or Q.

Data-oblivious algorithm: The above algorithm makes several computations over the tree to annotate the nodes of the SPQ tree with values b, b' and (x, y). Value b can easily be computed in the same manner as we computed the subgraph size in Section 7.3.2. Value b' of the left child is added only for parents of P nodes. When an Euler tour is going up the tree we can always check the value of $\mathsf{parent\_spq\_type}$ to know if b' of the left subgraph should be carried to the right one. Coordinates (x, y) for each node are computed from a small modification of the Euler tour: the left child needs know value $b(\triangle(G_2))$ and right child needs to know $b'(\triangle(G_1))$. It is easy to do this by always reading the next edge and remembering the last edge.

Given that we know the coordinates of each triangle, we can now assign coordinates for individual nodes. Recall that every leaf node of SPQ tree is associated with an edge while an internal node is either a DAG or a path of edges in the subtree rooted at this node. Hence, we can associate each internal node of SPQ tree, and edges in the corresponding Euler tour, with two nodes of the series-parallel graph that correspond to the source and the sink of the underlying subgraphs. Given a parent node of SPQ tree and source and sink nodes of its children, c_1 and c_2, if c_1^{sink} and c_2^{source} are equal then node c_1^{sink} is placed at $(c_1.x, c_1.y + c_1.b)$. Otherwise, we output a dummy.



(a) **(b)** **(c)**

Figure 7.4: (a) A series-parallel graph. (b) SPQ tree representation annotated with values b and b' (dashed rectangles). (c) The final drawing.

7.3.6 Drawing Trees with Bounding Rectangles

In this section, we present an algorithm that draws a binary tree T using a bounding rectangle approach from [10], adapted to the compress-scanning model. This algorithm is slightly different from the approaches we took in previous algorithms and involves a more complex way of converting it to fit data-oblivious mode. The original algorithm recursively assigns bounding rectangles to nodes of the tree. A leaf node is assigned a rectangle of size 2×1, while an internal node is assigned a rectangle that fits the bounding rectangles of its children. Each rectangle is represented by parameters width, height, and $\mathsf{refpoint}$ (left top corner). For a leaf, $\mathsf{width} = 2$. For an internal node, width is the sum of the widths of its children. The height of the rectangle is defined as $1 + \max_i \mathsf{child}_i.\mathsf{height}$. The bounding rectangle of the root has $\mathsf{refpoint} = (0, \mathsf{tree\_height})$. The $\mathsf{refpoint}$ of the ith child of node p is $(p.\mathsf{refpoint}.x + \sum_{j<i} \mathsf{child}_j.\mathsf{width}, p.\mathsf{height})$. A leaf node l is placed at point $(x, y) = (l.\mathsf{refpoint}.x + l.\mathsf{width}/2, l.\mathsf{refpoint}.y)$. An internal node is placed between its children, hence, a node l with children child_i $(i = 1, 2)$ is placed at point $(\sum \mathsf{child}_i.x/2, l.\mathsf{refpoint}.y)$.

Data-oblivious algorithm: An Euler tour over T allows to compute the width, level and $\mathsf{refpoint}$ values of the nodes. Computing the coordinates of an internal node requires knowing the coordinates of its children, and thus can be done only after the subtrees of both children are processed. If we use an Euler tour traversal, we need to store the coordinates of the points in the left subtree while processing the right subtree. We cannot store these coordinates in the private workspace since in the worst case their number is linear in the size of the tree. Indeed, in our previously described methods, we only store a constant number of values when traversing a tour. Therefore, in this section we propose a different technique that is based on a dashed-solid representation. This representation allows us to store only $O(\log n)$ coordinates in the worst case, which fits our compressed-scanning model.

In the *dashed-solid* representation of a tree [45], an edge $\textsf{parent-child}_i$ is said to be solid if $\textsf{parent.subsize}/2 < \textsf{child}_i\textsf{.subsize}$ and dashed otherwise. If the children have the same subsize, the right edge is solid and the left one is dashed. Thus, a parent node has a solid edge to only one of its children whose subtree has size equal or larger than that of the sibling. The main property of the dashed-solid assignment is that any path of the tree has $O(\log n)$ dashed edges. The dashed-solid representation can be computed from the subsize values using another Euler tour traversal (Section 7.3.2)

Given a dashed-solid representation, we compute the (x, y) coordinates by creating a tour around the tree where edges are accessed in a specific order. First, we go down the path of solid edges starting at the root. When a leaf is reached, we go back up until a node with a dashed edge is reached. We then recursively traverse the subtree connected to the dashed edge. To construct this traversal one needs to store with every node which one of its children is solid. The coordinates are computed as follows. We follow a solid edge path until a leaf l is reached and then the leaf node is assigned to coordinates $(l.\textsf{refpoint}.x + l.\textsf{width}/2, 0)$. We store these coordinates in variable s in private memory. When going up, if the parent node p does not have any other children, then we assign it to $(s.x, s.y)$ and continue traversing up the tree. If instead node p has a dashed edge to child c, then we recursively traverse the subtree of c, which results in the computation of the coordinates of c, denoted d. Once this traversal is finished, node p is assigned to coordinates $f = ((s.x + d.x)/2, 1 + \max(s.y, d.y))$. We now set $s = f$ and keep going up the solid path. Note that in the recursive traversal of the subtree of node c, we will store additional coordinates in private memory. Since a root-to-leaf path in the tree has no more than $\log n$ dashed edges, a private workspace of size $O(\log n)$ is enough to store all the coordinates needed by the traversal. (See Figure 7.5 for an example of the drawing.) We note that this algorithm can be extended to arbitrary trees if we

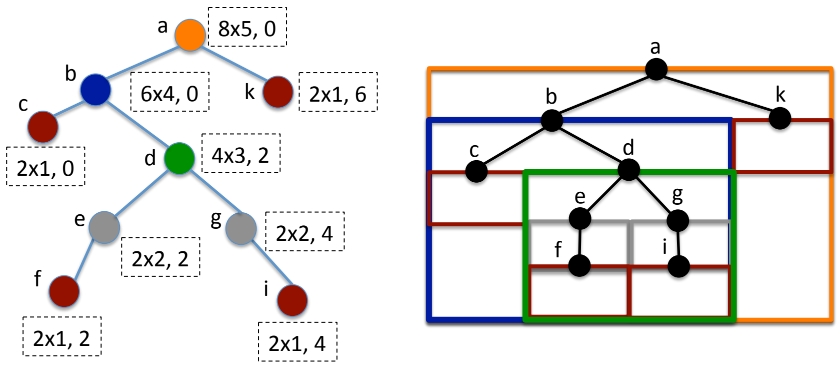


Figure 7.5: Example of the algorithm in Section 7.3.6. Tree annotated with the width, height and x coordinate of the top left corner of the bounding rectangle (refpoint). Figure on the right is the resulting drawing.

represent the dashed edges of a node as a balanced binary tree (see [10] for details).

7.3.7 Analysis

We have given drawing algorithms in the compressed-scanning model that consist of a constant number of Euler tours. In Section 7.3, we have shown that an Euler tour can be implemented with a single-round compressed scan, where, from the server's perspective, the items associated with the edges of the tour are accessed sequentially. Thus, the following theorem is a consequence of Theorem 25.

Theorem 26. *The drawing algorithms described in this section are data-oblivious according to Definition 3 and run in time $O(\mathsf{sort}_B(n))$ where n is the size of the input graph/tree and B is the oblivious sorting algorithm used between the rounds. Also, the private workspace has size $O(\log n + \mathsf{workspace}_B(n))$ for the bounding-rectangle tree-drawing algorithm and has size $O(\mathsf{workspace}_B(n))$ for the other algorithms.*

Our algorithms hide the combinatorial structure and layout of the graphs, while the number of edges and vertices is revealed. One can achieve even stronger privacy if dummy edges and nodes are added. From the point of view of the model, the input S is a larger set of elements and the running time of algorithm A increases as well.

Algorithm 8 Data-oblivious algorithm to compute a treemap drawing of an arbitrary tree.

```
out.node ← root, out.$P$ ← $[0,0]$, out.$Q$ ← $[w,h]$
write out
read $\pi(0)$ into $e$ {Get an edge corresponding to the leftmost edge from the root of $T$}
axis ← $0$, unit ← $w/e$.parent_area
{prevP, prevQ, unit, axis are maintained in private memory, $W$}
prevP ← $[0,0]$, prevQ ← $[w,h]$
while $e$.parent ≠ root and $e$.direction ≠ up do
  if $e$.direction = down then
    prevQ[axis] ← prevP[axis] + unit × $e$.child_area
    out.node ← child, out.$Q$ ← prevQ, out.$P$ ← prevP
    if $e$.child_outdeg $= 0$ and $e$.child_num $<$ $e$.parent_outdeg then
      prevP[axis] ← prevQ[axis] {Move the top left corner for the next child}
    else if $e$.child_outdeg $> 0$ then
      {Go further down the branch}
      unit ← (prevQ[$1 -$ axis] $-$ prevP[$1 -$ axis])$/e$.child_area
      axis ← $1 -$ axis
    end if
  else
    if $e$.child_num = $e$.parent_outdeg then
      {Going up again. Undo previous $P$, $Q$ changes.}
      branch_size ← unit × $e$.parent_area
      unit ← (prevQ[$1 -$ axis] $-$ prevP[$1 -$ axis])$/e$.parent_area
      prevP[axis] ← prevQ[axis] $-$ branch_size
      prevP[$1 -$ axis] ← prevQ[$1 -$ axis]
      axis ← $1 -$ axis
    end if
    out.node ← dummy, out.$Q$ ← $[0,0]$, out.$P$ ← $[0,0]$
  end if
  write out
  read $\pi(e$.tag$)$ into $e$
end while
Sort all output values by node field such that dummy values are in the end.
```

Chapter Eight

Conclusions

In this thesis we have developed several efficient and provably secure data-oblivious algorithms for privacy-preserving access to cloud storage. We have presented two schemes that obliviously simulate the access sequence of a general-purpose algorithm to memory of size n. In the first scheme, the $\log n$-hierarchical scheme, the client interacts with the server in the RAM model and has access to private memory of size n^ϵ, $0 < \epsilon < 1$. Each access is simulated obliviously with an amortized overhead of $O(\log n)$, with very high probability. The oblivious simulation is performed by random-looking accesses to a data structure based on hash tables and cuckoo hash tables that are stored at the server.

The second scheme is defined in the oblivious storage model. This model is arguably a more natural way to model interaction between the client and the cloud provider. The client and the server exchange messages of size $O(n^{1/c} \log n)$, $c \geq 2$, and the client can ask the server more complex queries than read and write. The construction uses our data-oblivious shuffle algorithm, the Melbourne shuffle, which is the first data-oblivious shuffle algorithm that is not based on sorting. This scheme incurs a constant amortized overhead over a non-oblivious access to the cloud storage.

We then have presented an optimization that is applicable to several oblivious access schemes, including our constructions. These schemes usually provide an efficient amortized access overhead but tend to have $\Omega(n)$ slowdown in the worst case. We have described a deamortization technique that makes worst case access overhead as efficient as the average case.

Finally, we have studied oblivious algorithms that hide the access pattern of specific applications instead of being general-purpose access simulations, such as above. We have introduced the compressed-scanning technique for designing data-oblivious algorithms in a cloud-computing environment. In a nutshell, this technique involves specifying an algorithm as a series of scans where data is processed using a small working storage. Using this technique, we have shown how to implement classic drawing algorithms for trees, series-parallel graphs, and planar st-digraphs (and variations of these algorithms) so that the client needs only a small amount of working storage (constant or logarithmic in the size of the data set) and can fully protect the privacy of the graph and its layout, beyond what can be accomplished by encryption alone.

Bibliography

[1] Miklós Ajtai. Oblivious rams without cryptogrpahic assumptions. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 181–190, New York, NY, USA, 2010. ACM.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[3] David Aldous and Persi Diaconis. Shuffling cards and stopping times. *The American Mathematical Monthly*, 93(5):333–348, 1986.

[4] Amazon S3 - The First Trillion Objects. Accessed in September, 2013. "`http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html`".

[5] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[6] Paola Bertolazzi, Robert F. Cohen, Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. How to draw a series-parallel digraph. *International Journal of Computational Geometry and Applications*, 4:385–402, 1994.

[7] Dan Boneh, David Mazières, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. Technical report, CSAIL, MIT, 2011. `http://dspace.mit.edu/handle/1721.1/62006`.

[8] David Cash, Alptekin Kp, and Daniel Wichs. Dynamic Proofs of Retrievability via Oblivious RAM. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 279–295. Springer Berlin Heidelberg, 2013.

[9] Vasek Chvátal. Lecture notes on the new AKS sorting network. Report DCS-TR-294, Computer Science Dept., Rutgers University, 1992. Available at `ftp://athos.rutgers.edu/pub/technical-reports/dcs-tr-294.ps.Z`.

[10] Robert F. Cohen, Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. Dynamic graph drawings: Trees, series-parallel digraphs, and planar *ST*-digraphs. *SIAM Journal on Computing*, 24(5):970–1001, 1995.

[11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *Proceedings of the 8th Conference on Theory of Cryptography*, TCC'11, pages 144–163, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing.* Prentice Hall, Upper Saddle River, NJ, 1999.

[13] Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete & Computational Geometry*, 7(4):381–401, 1992.

[14] Dropbox confirms security glitch–no password required. Accessed in September, 2013. "`http://news.cnet.com/8301-31921_3-20072755-281/dropbox-confirms-security-glitch-no-password-required/`".

[15] Emil Stefanov and Elaine Shi. Multi-Cloud Oblivious Storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 247–258, New York, NY, USA, 2013. ACM.

[16] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms (TALG)*, 6(4):66:1–66:19, 2010.

[17] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.

[18] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

[19] Oded Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques.* Cambridge University Press, 2001.

[20] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

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

[22] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[23] Michael T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–16. SIAM, 2010.

[24] Michael T. Goodrich and Michael Mitzenmacher. Anonymous card shuffling and its applications to parallel mixnets. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II*, ICALP'12, pages 549–560, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 95–100, New York, NY, USA, 2011. ACM.

[26] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM.

[27] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167. SIAM, 2012.

[28] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Graph drawing in the cloud: Privately visualizing relational data using small working storage. In Walter Didimo and Maurizio Patrignani, editors, *Graph Drawing*, volume 7704 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg, 2013.

[29] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '12, pages 513–524, New York, NY, USA, 2012. ACM.

[30] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization*, pages 284–291, 1991.

[31] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC Press, 2007.

[32] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: cuckoo hashing with a stash. *SIAM Journal on Computing*, 39:1543–1561, 2009.

[33] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 3rd edition, 1998.

[34] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156. SIAM, 2012.

[35] Steve Lu and Rafail Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. In Amit Sahai, editor, *Theory of Cryptography*, volume

7785 of *Lecture Notes in Computer Science*, pages 377–396. Springer Berlin Heidelberg, 2013.

[36] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer Berlin Heidelberg, 2013.

[37] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 311–324, New York, NY, USA, 2013. ACM.

[38] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *Proceedings of the 2009 European Symposium on Algorithms*, ESA'09, pages 1–10, 2009.

[39] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, New York, NY, USA, 2005.

[40] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The Melbourne Shuffle: Improving oblivious storage in the cloud. In *Proceedings of the 41th International Colloquium Conference on Automata, Languages, and Programming*, ICALP'14, 2014.

[41] Olga Ohrimenko, Michael T. Goodrich, Roberto Tamassia, and Eli Upfal. The Melbourne shuffle: Improving oblivious storage in the cloud. *CoRR*, abs/1402.5524, 2014.

[42] Rasmus Pagh and Flemming F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 52:122–144, 2004.

[43] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 571–582, New York, NY, USA, 2013. ACM.

[44] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, pages 197–214, 2011.

[45] Daniel Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–381, 1983.

[46] Emil Stefanov and Elaine Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy*, pages 253–267, 2013.

[47] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. In *Proceedings of the 2012 Network and Distributed System Security Symposium*, NDSS'12, 2012.

[48] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[49] Edward O. Thorp. Nonrandom shuffling with applications to the game of Faro. *Journal of the American Statistical Association*, 68(344):pp. 842–847, 1973.

[50] Peter Williams. Oblivious remote data access made practical. PhD thesis, Dept. of Computer Science, SUNY Stony Brook, 2012.

[51] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of the 2008 Network and Distributed System Security Symposium*, NDSS'08, 2008.

[52] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM.

[53] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 139–148, New York, NY, USA, 2008. ACM.

[54] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: a parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM.

[55] Xuan Zhou, HweeHwa Pang, and Kian-Lee Tan. Hiding data accesses in steganographic file system. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 572–, Washington, DC, USA, 2004. IEEE Computer Society.