DISSERTATION

SEARCHING OVER ENCRYPTED DATA

Submitted by

Tarik Moataz

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2017

Doctoral Committee:

Advisor: Indrajit Ray
Co-advisor: Indrakshi Ray

Frédéric Cuppens
Nora Boulahia Cuppens
Wang Haonan
Ross McConnell

ABSTRACT

SEARCHING OVER ENCRYPTED DATA

Cloud services offer reduced costs, elasticity and a promised unlimited managed storage space that attract many end-users. File sharing, collaborative platforms, email platforms, back-up servers and file storage are some of the services that set the *cloud* as an essential tool for every day use. Currently, most operating systems offer built-in outsourced cloud storage applications, by design, such as One Drive and iCloud, as natural substitutes succeeding to the local storage. However, many users, even those willing to use the aforementioned cloud services, remain reluctant towards fully adopting cloud outsourced storage and services. Concerns related to data confidentiality rise uncertainty for users maintaining sensitive information. There are many, recurrent, worldwide data breaches that led to the disclosure of users sensitive information. To name a few: a breach of Yahoo [47] late 2014 and publicly announced on September 2016, known as the largest data breach of Internet history, led to the disclosure of more than 500 millions user accounts; a breach of health insurers, Anthem [4] in February 2015 and Premera BlueCross BlueShield [13] in March 2015, that led to the disclosure of credit card information, bank account information, social security numbers, data income and more information for more than millions of customers and users. A traditional countermeasure for such devastating attacks consists of encrypting users data so that even if a security breach occurs, the attackers cannot get any information from the data. Unfortunately, this solution impedes most of cloud services, and in particular, *searching* on outsourced data.

Researchers therefore got interested in the following question: *how to search on outsourced encrypted data while preserving efficient communication, computation and storage overhead?* This question had several solutions, mostly based on cryptographic primitives, offering numerous security and efficiency guarantees. While this problem has been explicitly identified for more than a decade, many research dimensions remain unsolved.

The main goal of this thesis is to come up with practical constructions that are (1) suitable for real life deployments verifying necessary efficiency requirements, but also, (2) providing good security insurances. Throughout our research investigation, we identified symmetric searchable encryption (SSE) and oblivious RAM (ORAM) as the two potential and main cryptographic primitives candidate for real life settings. We have recognized several challenges and issues inherent to these constructions and provided a number of contributions that improve upon the state of the art.

First, we contributed to make SSE schemes more expressive by enabling boolean, semantic, and substring queries. Practitioners, however, need to be very careful about the provided balance between the security leakage and the degree of desired expressiveness. Second, we improve ORAM's bandwidth by introducing a novel recursive data structure and a new eviction procedure for the tree-based class of ORAM constructions, but also, we introduce the concept of *resizability* in ORAM which is a required feature for cloud storage elasticity.

ACKNOWLEDGEMENTS

First and foremost, this thesis is dedicated to my beloved *family*. My dear parents Hinda and Mustafa, unconditionally supporting me through these years, gave me the strength to go forward with their unselfish love and affection. This thesis is for the best brother in the world, Amine, who taught me not only how to solve my first equations, but also the importance of rigorousness in our scientific world. This dissertation is the result of many sacrifices made by my love Safâa, bearing with me all the research inconveniences, sharing my sorrow and happiness whenever a paper's acceptance notification pops up, always supporting me and encouraging me, this thesis is for her.

I am extremely grateful to my advisors Frédéric Cuppens, Indrajit Ray, Nora Cuppens-Boulahia and Indrakshi Ray for these wonderful years that we have spent together. Words alone cannot express my deep gratitude towards my advisors, for introducing me to the field of computer security, their insightfulness, trust, unconditional help and time spent working with me. I had the chance to be a part of this amazing joint PhD program between Telecom Bretagne, France and Colorado State University, USA that my advisors have carefully set up for me. This program gave me the opportunity to experience two different yet complementary research philosophies that I am sure is going to have an important impact on my research's vision. In addition to their flawless research guidance and valuable advices, my advisors grant me an enjoyable research freedom which had a considerable impact on my research's maturity. I want also to thank the members of my PhD committee, Ross McConnell and Wang Haonan, for all their valuable feedback, for their time and for being critical towards my research. I want to also thank the French PhD committee Seny Kamara and Joaquin Garcia-Alfaro for all their valuable comments.

Throughout the years I have spent for my PhD, I had the chance to work with amazing researchers who played an important role in my research orientation. I want to especially thank Erik-Oliver Blass for introducing me to the ORAM's field while at Northeastern University. Erik is one of the rare researchers with whom collaboration is not only fruitful but exiting, fun and enjoyable. I want also to thank Guevara Noubir, one of the smartest and humblest researchers I

have ever worked with. I want also to thank Travis Mayberry for all his advices and for walking me through the ORAM primitives.

My thoughts also go to my internship supervisor at Bell Labs, Abdullatif Shikfa. I am grateful to Abdullatif for introducing me to the applied cryptographic field and to the encrypted search area in particular.

I want to particularly thank Seny Kamara for playing a key role in greatly expanding my knowledge in the encrypted search and applied cryptography in general while interning at Microsoft Research but also during my current visit to Brown University. I am extremely indebted and words alone cannot express my deep gratitude.

Finally, I want to thank all my friends at both institutions and collaborators who supported me, encouraged me and trusted on me. My special thanks go to Ibrahim, Farid, Anis, Reda, Louis, Yaoqi and Shruti.

DEDICATION

To my parents

To my brother

To my love

TABLE OF CONTENTS

Abstract . . . . . **ii**

Acknowledgements . . . . . **iv**

Dedication . . . . . **vi**

List of Tables . . . . . **x**

List of Figures . . . . . **xi**

1 Introduction **1**

1.1 Research contributions 4

1.1.1 SSE contributions 4

1.1.2 ORAM contributions 7

1.2 Ongoing Work 9

1.3 Dissertation Outline. 11

2 Research Challenges and Background **12**

2.1 Research dimensions 12

2.1.1 Searchable Encryption 13

2.1.2 Oblivious RAM 18

2.2 Scheme Definitions 21

2.2.1 Setting definition 21

2.2.2 Structured Encryption (STE) 21

2.2.3 Tree-based Oblivious RAM 24

2.3 Security Definitions 26

2.3.1 STE security definition 29

2.3.2 ORAM security definition 30

2.4 Cryptographic Primitives 31

3 Related Works 33

3.1 Searchable Encryption 38

3.2 Oblivious RAM 40

4 Expressiveness in Symmetric Searchable Encryption 43

4.1 BSSE: Boolean SSE 44

4.1.1 Contribution Summary 44

4.1.2 BSSE construction 45

4.1.3 Security Analysis 51

4.1.4 BSSE Performance 52

4.2 3SE: Semantic SSE 54

4.2.1 Contribution Summary 54

4.2.2 Stemming Algorithms 55

4.2.3 Semantic SSE: Construction Overview 58

4.3 SED: Substring SSE 63

4.3.1 Contribution Summary 63

4.3.2 Substring search over Encrypted Data - Pre-construction 66

4.3.3 Substring Search Over Encrypted Data #1 – SED-1 66

4.3.4 Substring Search Over Encrypted Data #2 – SED-2 74

4.3.5 Security analysis for SED-2 77

4.3.6 SED-2 Generalization 82

4.3.7 Performance Analysis 84

5 Oblivious RAM 89

5.1 Resizable ORAM 89

5.1.1 Motivation and Findings 90

5.1.2 Resizable ORAM 91

5.1.3 Adding 92

5.1.4 Pruning 102

5.2 Recursive ORAM . . . 107
5.2.1 Contribution Summary . . . 108
5.2.2 Recursive Binary Trees . . . 109
5.2.3 $\mathcal{K}$-ary Trees . . . 119
5.2.4 Security Analysis . . . 120
5.2.5 Performance Analysis . . . 133
5.3 Constant bandwidth ORAM . . . 138
5.3.1 Contribution Summary . . . 139
5.3.2 Background: Onion ORAM . . . 140
5.3.3 Constant Communication ORAM . . . 143
5.3.4 C-ORAM analysis . . . 153
5.3.5 Evaluation . . . 164

6 Conclusion . . . 172

References . . . 173

LIST OF TABLES

4.1 Encrypted documents along with encrypted indexes in the outsourced server 47

5.1 Tree height comparison 134

5.2 Tree-based ORAM gain 134

5.3 Comparison of Onion ORAM and C-ORAM, containing block size, worst-case bandwidth, and number of homomorphic additions and multiplications. The simplified block value is a looser bound for easier comparison using $\lambda = \omega(\log N)$ and $\gamma = O(\lambda^3)$. 140

5.4 Bucket creation pattern in function of the eviction step. 171

LIST OF FIGURES

2.1 Setup phase: the user outsources all files in an encrypted form along with the secure searchable index. . . . 21
2.2 Query phase: (1) user sends an encrypted query that the server processes, (2) the server outputs the encrypted files without any knowledge of the query or files plaintext, and (3) the user decrypts the files. . . . 22

4.1 Search phase . . . 52
4.2 Orthogonal keywords construction . . . 52
4.3 Labels creation . . . 53
4.4 Semantic encrypted search 3SE. . . . 60
4.5 Stemming process . . . 61
4.6 Letter Frequency . . . 73
4.7 Length Frequency . . . 74
4.8 Substring search over encrypted data SED − 2 for the wildcard search case . . . 76
4.9 Time of inverted index encryption . . . 86
4.10 Number of unique keywords . . . 86
4.11 Searchable part construction . . . 87
4.12 Size of the encrypted data structure . . . 87
4.13 Search phase time . . . 87

5.1 Communication, blocks per access . . . 101
5.2 Storage cost, blocks . . . 101
5.3 Illustration of permute-and-merge process. Bucket (2) is permuted and then merged with bucket (1) to create a new, combined bucket (3). . . . 106
5.4 Structure of an r-ORAM . . . 110
5.5 r-ORAM path length distribution . . . 116
5.6 r-ORAM Map addressing . . . 118

5.7 Structure of an r-ORAM . . . 123
5.8 Communication per access . . . 136
5.9 Communication cost per 100 accesses . . . 136
5.10 Average height comparison . . . 136
5.11 Communication per access . . . 136
5.12 Stash size, $z = 6$, number of operations in $\log_2$ . . . 138
5.13 Average stash size, operations in $\log_2$ . . . 138
5.14 C-ORAM bucket structure . . . 146
5.15 Buckets on evicted path are with horizontal hatching. Bucket 3 is a copy of the root. Bucket 4 results from merging buckets 1 and 2. Bucket 6 results from merging 1, 2, and 5. . . . 154
5.16 Illustration of nine evictions. Numbers below leaves represent the order of reverse deterministic lexicographic eviction. Buckets with same shapes were full and then evicted at the same step. Example: buckets with triangular shape are evicted in step 9. . . . 155
5.17 Minimum efficient block size for C-ORAM and Onion ORAM . . . 164
5.18 Required bucket size in relation to security parameter . . . 164
5.19 Required ciphertext operations for one access . . . 165
5.20 Comparison of computation time for one access . . . 165

Chapter 1

Introduction

> Truth is ever to be found in simplicity, and not in the multiplicity and confusion of things.
>
> *Isaac Newton*

We are in an era where outsourcing data to cloud servers becomes a must for most end users. The amount of data generated by nowadays services grows sharply in a daily basis. Managing local infrastructures in this case becomes a concrete burden for end users and companies. Moreover, local infrastructures can be even considered as a usability hindrance for many services such as email servers [15], file sharing systems [5], and collaborative platforms [7]. Fortunately, the recent widely adoption of outsourced infrastructures, data centers and externalized computing services such as Amazon Web Serivce [3], Microsoft Azure [12], Google Drive [8], IBM cloud [10], HP public cloud [9], Dropbox [6] and so on, makes cloud services and infrastructures more appealing than ever. In particular, cloud computing brings several novel opportunities to the end users with a promised unlimited amount of managed storage space, and an ubiquitous access to the infrastructures. With very competitive prices, companies are tending to outsource many of their services and locally stored data to the outsourced servers. However, while outsourced infrastructures come with several advantages, an extremely important issue remains unsolved. Cloud infrastructures have a serious issue when dealing with users' sensitive data, in particular, preserving cloud services while users' data must be kept private. In an outsourced setting, end users and companies do not necessarily trust the servers and even often consider cloud servers as *untrusted* and *dishonest*. In such a setting, the server is a potential adversary that can maliciously look after users' files, documents and pictures and therefore gather information that are a-priori private. In the meantime, one of the major cloud services consists of performing search operations over outsourced data. In this case, an entity, either an end user or a company, outsources its local data to an external server to perform search operations when required. Search can be considered as one of the most fundamental oper-

ations and can be used as a building block to simulate more complex operations and algorithms. One straightforward and naive solution that preserves user's confidentiality would be to encrypt all of the data before outsourcing it to the cloud. However, current encryption schemes randomize data such that a search operation becomes almost impossible over the encrypted data. That is, to search for a keyword or a picture with a specific feature, the client needs to download the entire data stored on the outsourced servers, decrypt it before performing a local search. This is clearly unfeasible and will not scale to real life dataset sizes as it needs a large local client memory, a high-speed bandwidth, and an extremely efficient end user's commodity machine to run cryptographic decryption operations. These assumptions are clearly contradictory to the motivation behind a prior migration to cloud infrastructures. Researcher therefore got interested in the following question:

> *How can we search on encrypted data while preserving realistic communication, computation and storage overhead?*

This question has been *explicitly* answered for the first time by Song, Wagner and Perrig [130] who have presented the first construction that can enable searching over encrypted data. This construction is known as *symmetric searchable encryption* primitive (SSE). Concretely, searchable encryption is a cryptographic primitive that allows the user to search for any textual keyword in an encrypted dataset using encrypted queries. The cloud server, using the SSE protocol, can search on the encrypted dataset without decrypting the dataset or the query, and while being able to provide correct encrypted results. Researchers have then worked towards enhancing SSE protocols along many dimensions such as decreasing the search overhead by reducing the communication and search complexities, as well as reducing the storage overhead, and strengthening the security guarantees.

SSE was the first dedicated primitive to explicitly solve the encrypted search problem. Nevertheless, the same problem can be solved by leveraging many previously existing cryptographic primitives such as oblivious RAM (ORAM) and private information retrieval (PIR). ORAM [70] was introduced late 80s by Goldreich and Ostrovsky, while PIR [45] was introduced mid 90s by Chor, Goldreich, Kushilevitz and Sudan. ORAM, when first introduced, was designed as a prevention tool against retro-engineering. An entity that has an access to the memory can infer

incredible amount of information by just observing the access pattern made by the CPU into the memory. Also, ORAM, when first introduced, was extremely inefficient, theoretically minded and was therefore dismissed from most practical settings. Recently in 2011, Shi et al. [129] introduced a new way to design ORAM schemes in such a way that the asymptotics become sub-linear in the worst-case, instead of linear, with a smaller hidden constant in the big-O notation. Shi et al. introduced the first practically feasible ORAM, and since, ORAM has been considered as a potential solution for the the encrypted search problem. Conceptually, ORAM is very different from SSE in many aspects that will become clearer later on in Chapter 2. A high level difference to keep in mind between SSE and ORAM consists of their security guarantees and efficiency. ORAM offers much better security insurances when compared to SSE, at the cost of being less efficient.

Private information retrieval is another candidate to the encrypted problem. A user can hide her *search pattern* even if the outsourced data is not encrypted. PIR somehow offers better security guarantees and can be considered more powerful than SSE and ORAM as there is no need to encrypt the data. However this setting is different from our encrypted search problem where keeping data private is as (if not more) important as (than) hiding the search pattern. Fortunately, PIR can be easily adapted to a client / server setting and data can be also encrypted. PIR comes with two different instantiations: (1) a first instantiation that only works in multiple server setting, dubbed IT-PIR for information-theoretic PIR. In this case, data needs to be replicated over non-colluding multiple servers, (2) a second instantiation, more adapted to our client / server setting, is a single server computational PIR, dubbed C-PIR. At a higher level, PIR offers similar security insurance if compared to ORAM, however, with its linear computational time, PIR is definitely not suitable for search purposes.

Throughout our research, it becomes clear to us that SSE and ORAM are two potential candidates to search over encrypted data. Our choice to focus on these two primitives is twofold: first, both of the primitives offer at least sub-linear search complexities either in terms of communication or computation on the server side. This features makes ORAM and (especially) SSE suitable for very large data sets. Second, both of these primitives offer interesting security insurance with a very well understood leakage profile. SSE and ORAM, as solutions for the encrypted search

problem, can be suitable for two different settings: first, users who are lenient with some leakage related to their private data, but require very efficient and scalable protocol, and second, users who look after better security insurances, while potentially willing to give away somehow the scheme efficiency. Note however, that both of these protocols can perform search operations in a sub-linear time with different complexities that we are going to detail later on in Chapters 2 and 3.

1.1 Research contributions

For this thesis, we have been interested in solving *the efficient and expressive encrypted search problem*. For this purpose, we have focused on improving both SSE and ORAM primitives along several dimensions. Our ultimate goal was to build schemes that are *practical*, *efficient* and *suitable for a real-life deployment*, while preserving constructions security insurances. On a first hand, we proposed several SSE constructions that have enhanced the search expressiveness such as proposing one of the first instantiations of semantic, boolean and substring SSE schemes, as well as improving the leakage of conjunctive search. We have been also interested in applying some of the SSE techniques on some of the real worlds settings. On the other hand, we have also presented different improvements over the ORAM literature, especially, presented one of the most efficient constructions in terms of communication overhead and introducing the concept of resizability in ORAM literature. In the following, I will briefly present our thesis's contributions and defer greater details to the upcoming chapters, namely, Chapters 4 and 5. Please note that part of our contributions' descriptions in the following might sound technical and require some preliminaries and technical background that will be later introduced in Chapter 2.

1.1.1 SSE contributions

Boolean SSE[1]**.** BSSE was among the first candidates to introduce boolean SSE in literature. Departing from many previous works that focused on queries consisting of a single keyword [37, 42, 50, 66, 130], we consider in this work the case of queries consisting of arbitrary boolean expressions on keywords, that is to say conjunctions and disjunctions of keywords and their complement.

[1]This work has been published as a full paper in ACM ASIACCS 2013, as a poster in Financial Cryptography 2013, and has been a subject to an European and US patent [110].

Our construction of boolean symmetric searchable encryption BSSE is mainly based on the orthogonalization of the keyword field according to the Gram-Schmidt process. Each document stored in an outsourced server is associated with a label which contains all the keywords corresponding to the document, and searches are performed by way of a simple inner product. BSSE's search complexity is in $O(n)$ where n is the number of documents stored in the outsourced server. We will detail this construction in Chapter 4, Section 4.1.

Semantic SSE[2]. Current SSE constructions lack searching capabilities beyond exact search on single keywords and are thus less expressive than plaintext algorithms that take into account the meaning of the query and return results corresponding to semantically close keywords. We thus advocate for the need of semantic search over encrypted data. We study the state of the art in stemming algorithms and in searchable encryption and propose, 3SE, the first semantic symmetric searchable encryption construction. This improvement over searchable encryption has similar security as the prior work of Curtmola et al. [50] on symmetric searchable encryption as well as being optimal in terms of computational overhead. This construction can be further enhanced with better stemming algorithms. Furthermore, it can be easily plugged in more expressive construction such as the one by BSSE, OXT [38, 110] or more recently IEX [82]. We will detail 3SE in Chapter 4, Section 4.2.

Privacy-preserving conjunctive SSE[3]. Passive cloud servers can perform inference attacks to recover data or queries based on the query leakage. In this work, we propose two gradually improved privacy-preserving conjunctive symmetric searchable encryption (PCSSE) schemes which allow cloud servers to perform conjunctive keyword searches on encrypted documents with different privacy assurances. Our scheme generates randomized search queries to randomize the search pattern. PCSSE is also able to hide the number of keywords in a query and also the number of keywords contained in an encrypted document to a publicly known upper bound. However, the main downside of this work is its linear search time which makes it very complicated to scale for

[2]This work has been published in IEEE ICT 2013 and has been a subject to an European patent [111].
[3]This work has been published as a full paper in IFIP DBSec 2014 [107].

large data set. This work targets small to medium data sets. Moreover, hiding the search pattern while leaking the access pattern, for some cases, might be meaningless.

SSE in Real World[4]. Aside from focusing on enhancing SSE constructions, we have also been interested in incorporating these techniques in different realistic settings, in particular, in secure biometric storage settings, as well as in medical health databases. Here are two brief descriptions of our two settings:

- Modern day biometric systems, such as those used by governments to issue biometric-based identity cards, maintain a deterministic link between the identity of the user and her biometric information. However, such a link brings in serious privacy concerns for the individual. Sensitive information about the individual can be retrieved from the database by using her biometric information. Individuals, for reasons of privacy therefore, may not want such a link to be maintained. Deleting the link, on the other hand, is not feasible because the information is used for purposes of identification or issuing of identity cards. In this work, we address this dilemma by hiding the biometrics information, and keeping the association between biometric information and identity probabilistic. We extend traditional Bloom filters to store actual information and propose a new SOBER data structure for this purpose. Simultaneously, we address the challenge of verifying an individual under the multitude of traits assumption, so as to guarantee that impersonation is always detected. We discuss real-world impersonation use cases, analyze the privacy limits, and compare our scheme to existing solutions.

- In this work, we present a novel scheme that allows multiple data publishers that continuously generate new data and periodically update existing data, to share sensitive individual records with multiple data subscribers while protecting the privacy of their clients. An example of such sharing is that of health care providers sharing patients' records with clinical researchers. Traditionally, such sharing is performed by sanitizing personally identifying information from individual records. However, removing identifying information prevents any

[4]Two research works published as full papers in IFIP DBSec 2014 and 2015 [92, 106].

updates to the source information to be easily propagated to the sanitized records, or sanitized records belonging to the same client to be linked together. We solve this problem by utilizing the services of a third party. The scheme is based on single keyword SSE primitives and elliptic curve El-Gamal that do not require shared encryption keys between the parties.

1.1.2 ORAM contributions

Resizable ORAM[5] Although newly proposed, tree-based Oblivious RAM schemes are drastically more efficient than older techniques, they come with a significant drawback: an inherent dependence on a fixed-size database. Yet, a flexible storage is vital for real-world use of Oblivious RAM since one of its most promising deployment scenarios is for cloud storage, where scalability and elasticity are crucial. We revisit the original construction by Shi et al. [129] and propose several ways to support both increasing and decreasing the ORAMs size with sub-linear communication. We show that increasing the capacity can be accomplished by adding leaf nodes to the tree, but that it must be done carefully in order to preserve the probabilistic integrity of data structures. We also provide new, tighter bounds for the size of interior and leaf nodes in the scheme, saving bandwidth and storage over previous constructions. We will detail our solution in Chapter 5, Section 5.1.

Recursive ORAM[6] We present a new, general data structure that reduces the communication cost of recent tree-based ORAMs. Contrary to ORAM trees with constant height and path lengths, our new construction r-ORAM allows for trees with varying shorter path length. Accessing an element in the ORAM tree results in different communication costs depending on the location of the element. The main idea behind r-ORAM is a recursive ORAM tree structure, where nodes in the tree are roots of other trees. While this approach results in a worst-case access cost (tree height) at most as any recent tree-based ORAM, we show that the average cost saving is around 35 % for recent binary tree ORAMs. Besides reducing communication cost, r-ORAM also reduces storage overhead on the server by 4% to 20% depending on the ORAMs client memory type. To

[5]This paper has been published in Financial Cryptography and Data Security 2015 [109]. This works was done as a research visiting student to Northeastern University.

[6]This paper has been published as a full paper in the journal of PoPETS 2015 [105]. This works was done partially as a research visiting student to Northeastern University.

prove r-ORAMs soundness, we conduct a detailed overflow analysis. r-ORAMs recursive approach is general in that it can be applied to all recent tree ORAMs, both constant and poly-log client memory ORAMs. Finally, we implement and benchmark r-ORAM in a practical setting to back up our theoretical claims. We present r-ORAM in Chapter 5, Section 5.2.

Constant communication ORAM[7]. There have been several attempts recently at using homomorphic encryption to increase the efficiency of Oblivious RAM protocols. One of the most successful has been Onion ORAM [56], which achieves $O(1)$ communication overhead with polylogarithmic server computation. However, it has two drawbacks. It requires a large block size of $B = \Omega(\log^6 N)$ with large constants. Moreover, while it only needs polylogarithmic computation complexity, that computation consists mostly of expensive homomorphic multiplications. In this work, we address these problems and reduce the required block size to $B = \Omega(\log^4 N)$. We remove most of the homomorphic multiplications while maintaining $O(1)$ communication complexity. Our idea is to replace their homomorphic eviction routine with a new, much cheaper permute-and-merge eviction which eliminates homomorphic multiplications and maintains the same level of security. In turn, this removes the need for layered encryption that Onion ORAM relies on and reduces both the minimum block size and server computation. We detail C-ORAM in Chapter 5, Section 5.3.

OblivP2P: a distributed ORAM for Peer-to-Peer networks[8]. Peer-to-peer (P2P) systems are predominantly used to distribute trust, increase availability and improve performance. A number of content-sharing P2P systems, for file-sharing applications (e.g., BitTorrent and Storj) and more recent peer-assisted CDNs (e.g., Akamai Netsession), are finding wide deployment. A major security concern with content-sharing P2P systems is the risk of long-term traffic analysis — a widely accepted challenge with few known solutions.

[7]This paper has been published as a full paper in ACM CCS 2015 [108]. This works was done partially while visiting Airbus Group of Innovation.

[8]This paper has been published as a full paper in USENIX 2016 [80]. This works was done partially while visiting National University of Singapore.

As our result, we present a new approach to protecting against persistent, global traffic analysis in P2P content-sharing systems. Our approach advocates for hiding data access patterns, making P2P systems oblivious. We propose OblivP2P — a construction for a scalable distributed Oblivious RAM protocol, usable in a real P2P setting. OblivP2P leverages recent advances in tree-based ORAM and PIR to better distribute bandwidth and computation on the entire network. Our experimental results show that OblivP2P linearly scale with the number of peers in the network with seven 512 KB requests per seconds.

1.2 Ongoing Work

In this section, we present some of our works that are currently investigated. These works fall within the same line of research and present factual improvements over the state of art of both SSE and ORAM. In particular, we present further enhancements of SSE constructions by proposing the first non-interactive worst-case sub-linear dynamic boolean SSE [82], a substring SSE and we propose the first formal study to encrypted databases [84][9]. In addition, we further reduce ORAM communication overhead by introducing constant communication ORAM under the multiple-server setting. In the following, we briefly introduce the ongoing works:

Optimal Boolean SSE. Previous works on boolean SSE offer either a linear worst case computation with optimal communication [38, 110], or sub-linear computation with non-optimal communication overhead while being interactive [60, 117]. In this work, we propose, IEX, the first work that meets both requirements by providing optimal computation time with optimal communication. IEX can be extended to handle dynamic scenarios while providing forward security. This work starts by showing that disjunctive queries can be efficiently solved based on the inclusion-exclusion paradigm that transforms any disjunction to a set of conjunctions. Intuitively, as conjunctions have been efficiently solved in literature [38], disjunctions can also be solved efficiently given such a transformation. Moreover, we show how to extend disjunctive SSE to a boolean SSE. As a building block, we also introduce a new highly compact SSE construction. Finally, we imple-

[9] Work done while at Microsoft Research and Brown University.

ment a new encrypted search framework, Clusion [83] that implements all our IEX instantiations and therefore backs up our theoretical findings.

Substring SSE. We propose SED a general solution to the problem of efficient substring search over encrypted data. The solution enhances existing single keyword SSE constructions by allowing searching for any part of encrypted keywords without requiring one to store all possible combinations of substrings from a given dictionary. The proposed technique is based on the idea of letter orthogonalization that allows testing of string membership by performing efficient inner products. SED is adaptively secure and we provide details about our leakage profile. The protocol is also efficient in that the search complexity is linear in the size of the keyword's universe. We finally run several experiments on a sizeable real world dataset to evaluate the performance of our protocol.

Oblivious Substring Search Expressiveness in SEE constructions often comes at the cost of more leakage. Every query will leak more information to the server. Recent SSE cryptanalysis works [36, 113, 142] show that leakage in SSE can lead to the disclosure of users' queries. In light of these works, we propose an oblivious substring search protocol. While any data structure can be stored as an ORAM, we demonstrate that a dedicated construction based on suffix arrays and suffix trees can reduce the ORAM overhead significantly. Our scheme, OSS, leaks only the length of the query which is much lesser if compared to SED or to recently introduced substring SSE constructions such as [44, 59].

Constant Communication ORAM with no encryption. Recent techniques reduce ORAM communication complexity down to constant in the number of blocks N [56, 108]. However, they induce expensive additively homomorphic encryption on both the server and the client. As an alternative, we present an information-theoretically secure ORAM which, at the expense of requiring multiple servers, allows for substantially reduced client and server computation being constant in N. In essence, our idea is to combine ORAM with Private Information Storage. By assuming a small number of non-colluding servers, we show how homomorphic encryption can be replaced with much simpler XOR operations. Besides $O(1)$ communication complexity, our construction

also features $O(1)$ client memory complexity and a reasonable block size of $\Omega(\log^4 N)$. This leads to an ORAM which is extremely lightweight and suitable for deployment even on resource-constrained devices.

Encrypted Relational Database In this work, we show how to encrypt a relational database in such a way that it can efficiently support a large class of SQL queries. Our construction is based solely on structured encryption [42] and does not make use of any property-preserving encryption (PPE) schemes such as deterministic and order-preserving encryption. As such, our approach leaks considerably less than PPE-based solutions which have recently been shown to reveal a lot of information in certain settings [113]. Our construction achieves asymptotically optimal query complexity under very natural conditions on the database and queries.

1.3 Dissertation Outline.

In Chapter 2, we detail the main research challenges in the encrypted search problem, we then introduce some necessary technical background. In particular, we formalize the search scenario, detail SSE primitive, its generalization to structured encryption (STE), and tree-based ORAM construction. Moreover, we introduce some security definitions for both STE and ORAM protocols. In Chapter 3, we provide a detailed state of the art of STE and ORAM. In the same chapter, we also recall some cryptographic primitives that can be also used for the encrypted search problem. In Chapter 4, we present boolean, semantic and substring SSE constructions as our major findings in SSE. Similarly, in Chapter 5, we detail recursive, constant and resizable ORAM as our major findings in ORAM. In Chapter 6, we conclude.

Chapter 2

Research Challenges and Background

Learning never exhausts the mind.

Leonardo da Vinci

In this chapter, we start by presenting a global overview of the encrypted search realm. We will particularly be interested in listing the main challenges in this field and the key research questions that the community is still being interested to. In particular, we will focus on symmetric searchable encryption and oblivious RAM, and we list several research dimensions that we have identified as important to the community and that we have investigated in part. Then, we present some necessary preliminaries for a better understanding of our technical contributions. We first detail the setting of searching over encrypted data. Second, we introduce a formal definition of structured encryption (a generalization of symmetric searchable encryption) and oblivious RAM along with a general framework for existing tree-based ORAM instantiations. Third, we detail the security definition for searching over encrypted data with an emphasis on STE and ORAM cases. Finally, we recall some necessary preliminary cryptographic primitives such as private key cryptosystems, pseudo-random function (PRF) and generators (PRG). We want to point out that this section is not exhaustive and more detailed preliminaries are going to be introduced when required throughout the paper.

2.1 Research dimensions

Searching over encrypted data is an end-to-end process between a client(s) and a server(s). The aim of such protocol is to outsource data to an untrusted server while preserving both client's confidentiality and search functionality. The process of searching over encrypted data starts with an off-line phase of data processing, called *setup phase*, conceptually very similar to plaintext data pre-processing. Pre-processing the entire user's data is required to create a searchable form of the data, called secure searchable index or encrypted data structure. The resulting pre-processed

encrypted data structure helps to greatly speed-up the search to reach optimal search times. Encrypted data structures can have different forms and structures based on the client needs. As an instance, a data structure can be further optimized to handle faster conjunctive or boolean searches at the cost of further pre-processing time. Finally, the client encrypts the documents that are going to be outsourced along with the encrypted data structure. The second phase, that we refer to as *search phase*, starts when a client desires to query the server. For this, she constructs an encrypted query that the server processes without any knowledge of its content, runs the query on the encrypted data structure that will output the necessary pointers to retrieve the files. The server will finally send the corresponding encrypted files to the client who is going to decrypt to retrieve the corresponding plaintext files.

In our research work, our research scope focuses on two cryptographic primitives that solve the encrypted search problem. First, symmetric searchable encryption (SSE), later generalized to structured encryption [42], and second Oblivious RAM (ORAM). Each of these two primitives has its proper challenges occurring either during the setup or the search phase. In the following, we give more details about these challenges that were identified throughout our technical research studies.

2.1.1 Searchable Encryption

Symmetric searchable encryption (SSE), or its generalization to structured encryption (STE)[1], is a tuple of three probabilistic algorithms running in a polynomial time: Setup, Lookup and Search. At a higher level, Setup is an algorithm that takes the plaintext data set as input, and outputs the encrypted data structure along with the encrypted files. Lookup (called also Token) is an algorithm that takes as input a boolean function of keywords and outputs a secure token. Search (called also Query) is an algorithm that takes as input both the secure token and the encrypted data structure, and outputs the desired encrypted files. We give a more formal definition of SSE in Section 2.2.2. From a research perspective, there are many challenges and balances inherent to each of these algorithms. The setup overhead, expressiveness of the query, computation and communication

[1] In the remaining sections of the dissertation we use STE and SSE interchangeably.

overhead, I/O efficiency and leakage (security) are fundamental challenges for SSE constructions. In the following, we give more details about each of these challenges.

2.1.1.1 Expressiveness of the Query

Expressiveness in SSE englobes the different types of queries that an encrypted structure can be queried on (or support). Query expressiveness includes, but not limited, to conjunction, disjunction, and boolean queries. Recently, many researchers got interested in building encrypted structures for different types of queries such as range [59], similarity [22, 136], shortest distance in graphs [42, 104], and semantic queries [111]. From a theoretical perspective, all queries can be represented as a boolean expression. As an instance, an integer range query to find all files containing a value within the interval $[a, b]$ is equivalent to a conjunctive query composed of all integers within the interval such that $\{a, a + 1, \cdots, b - 1, b\}$. Enhancing query expressiveness has significant impact on security and efficiency. Naively, any complex query can be constructed while leveraging single keyword SSE. In general, boolean queries can be represented in a CNF or DNF formulas [2], and therefore multiple keywords within the boolean query can be searched for in an atomic manner i.e., by invoking single keyword SSE over every term within the boolean query. For instance, if the query equals w_1 AND w_2, then one can separate the query into three sub-queries such that the client runs as a sub-routine a single keyword SSE protocol on w_1, then redoes the same process on w_2. The client gets two sets containing all encrypted files matching respectively keywords w_1 and w_2, separately. The client finally outputs the intersection of both of these sets. While this simple solution provides correctness, it has some non-trivial efficiency downsides. First, if we consider a worst-case scenario where w_1 and w_2 match more or less the entire encrypted data set, then the client might have to download the entire encrypted data set where the intersection only consists of a couple of encrypted files. Clearly, this can have a devastating impact on communication bandwidth between the client and the server. One however, can work with an SSE that reveals the answer, and therefore the server will perform the intersection instead of the client, but this would have an important implication on the security of the construction as it

[2] CNF stands for conjunctive normal form, while DNF for disjunctive normal form.

leaks more than needed. As the server being traditionally considered as the main adversarial entity in data externalization, one needs to reduce as much as possible the amount of information that can be leaked to the server during the search phase. In particular, fetching documents for every query can lead to disclosing unnecessary information to the server[3]. The server can infer, for instance, the number of documents that an intersection of any two disjoint keywords in the query contains. An adversary with some auxiliary information can infer more information. Reducing leakage is extremely important as it has recently been shown that it can be used to recover plaintext queries or the outsourced data itself by [36, 78, 113]. Ideally, a conjunctive query in an SSE construction only consists of *one* encrypted query that the server processes without disclosing which keyword in the conjunction matches which encrypted file. That is, an ideal leakage in a conjunctive SSE only equals the final result of the query.

To sum up, expressiveness comes up with many security and efficiency challenges. The ultimate goal is to reduce leakage while not impacting efficiency. Finding out the right balance greatly depends on the underlying application.

2.1.1.2 Setup overhead

To securely outsource a data set, the user needs to generate an encrypted data structure (or encrypted searchable indexes). The Setup algorithm is an off-line step that is performed once in the lifetime of the system. It takes as input the plaintext data set, and involves different operations depending on the degree of expressiveness desired for the STE construction. These operations generally include data parsing, stemming operations, evaluating cryptographic primitive operations. In the case of single keyword search, these operations are at least linear in the size of the data set. However for more expressiveness, the setup can be polynomial in the size of the data set. This can take a considerable amount of time especially for large data set. Researchers [38] reported that it can take days of processing time using *powerful* servers to generate the encrypted data structure. Moreover, the size of the encrypted data structure often equals the size of the entire data

[3]This is only true when considering a response-revealing STE construction. That is, the query algorithm outputs the answers in plaintexts. Response-hiding STE constructions leak much less and will be considered as optimal solutions for boolean queries with respect to leakage. We will details these two types of STE in Section 2.2.2

set which represents also an additional storage cost. For average users with a normal commodity machine, the setup represents a considerable one-time cost and can take much more time than the one reported so far by the aforementioned research papers.

Some STE constructions can enable the client to perform the setup operation in an on-line manner, i.e, generate the encrypted data structure in a step-by-step manner. This setup is possible if the STE is a *forward-secure dynamic* scheme, or only *dynamic* but under the constraint that the client must not perform any search operation as long as the setup did not terminate yet. Forward secrecy is a property that allows the client to unlink updates from previous search queries. That is, an adversary cannot infer any information from the updates knowing the search tokens.

Another research challenge in the setup phase is the creation of the encrypted data structure that preserves the same search efficiency whether the structure is stored in the main memory or in the hard disk. This research challenge is known in literature as the *locality* challenge. The security of the encrypted data structure consists in part of scrambling data such that it will be uniformly distributed. That is, when the encrypted data structure is stored in the hard drive, accessing non-contiguous memory blocks results in considerable delays. Consequently, the search becomes extremely slow when compared to the same search if the structure is kept in the main memory. Recent results show that the most efficient schemes in theory are not necessarily the best in practice in case of large data sets [40].

2.1.1.3 Computation overhead

Computation overhead is one of the main comparison metric in SSE constructions. When SSE was first proposed in literature [130], the search complexity was linear in the size of the entire data set. In particular, if we consider that all n files are textual, and each has size m, then the search can be performed in $O(m \cdot n)$ for single keyword search. Four years later, Goh [66][4] succeeded to reduce the search overhead to be only linear in the number of stored documents. Moreover, the report introduced the first index based SSE and proposed further improvements that were in the appendices that can further reduce the computation to be only logarithmic in n. In 2006, Curtmola

[4]The report titled *Secure Indexes* surprisingly has never been published

et al. [50] presented optimal search complexity construction, i.e., the computation performed by the server is only linear in the result set. While computation overhead has been more or less solved for single keyword search, for more expressive constructions it is not. Boolean SSE schemes are still not achieving optimal search efficiency and represents an on-going challenge in the SSE community. The difficulty behind this challenge consists of finding the right balance between storage, security and search efficiency.

2.1.1.4 Storage overhead

Storage overhead varies depending on the expressiveness of the construction as well as the desired level of security. It is very challenging to assert that an STE scheme has the best storage overhead while neglecting the level of security it offers. Reducing storage overhead of the encrypted data structure is also another research dimension of SSE constructions. As an instance, there are many expressive SSE constructions that can require exponential storage for optimal search computation, but clearly such solutions are not realistic. In general, for single keyword SSE, a widely acceptable storage complexity is in $O(n)$, which translates to be the total size of the entire data set. If a user has, for example, 10 GByte of files to store, then the encrypted data structure will have a size of 1 GByte, as an instance. The storage overhead of single keyword SSEs has been recently reduced by Cash et al. [37]. To sum up, it is very challenging to find the right balance between computational overhead and storage overhead.

2.1.1.5 Security

Security in SSE, when first proposed, was similar to the security of secret key encryption scheme, i.e., the SSE scheme is secure if data stored in the server is indistinguishable from randomly generated data. This security definition is very limited and not really adapted to SSE schemes where the protocol is interactive. In SSE, the knowledge of the server (adversary) increases with the search queries that the client evaluates. For example, the server can infer how many files are retrieved for every query, can know whether a query is repeated and so on. These kind of information were not captured by the first SSE security model. Curtmola et al. [50] proposed the first security definition that captures these information in the form of stateful *leakage*

functions. The authors show the first construction that results in a *semantic* secure[5] construction that will leak two types of leakage: a *setup* leakage and a *query* leakage[6]. At a higher level, this underlines that the proposed construction will not leak anything else other than these two leakages. The former leakage often captures information disclosed to the server by just looking at the encrypted data structure. It is usually the case for secure SSE constructions that the setup leakage only consists of the maximum size of data set [42]. The query leakage consists of the *search pattern* and *access pattern* that capture the information leaked during the search phase. These two leakages record, for example, all file identifiers, pointers, and queries that have been accessed or issued during a search phase. These two leakages can contain more information depending on many parameters such as the expressiveness, storage and computation overhead of the SSE construction.

While providing a formal leakage description is very important for every SSE construction, reducing this leakage is also very valuable. There were recently many attempts to show that an adversary can recover the plaintext of queries [36, 78, 113, 142] if some auxiliary information is obtained. These works will help the community to better understand the impact of leakage of STE constructions.

2.1.2 Oblivious RAM

Oblivious RAM (ORAM) [70] is a cryptographic primitive that hides the access pattern of a trusted CPU to an untrusted memory. From an untrusted memory perspective, any two accesses to the memory are indistinguishable even if the trusted CPU is accessing the same data. ORAM has many applications but we will only focus on secure data externalization. In this context, the trusted CPU can be pictured as the client and the untrusted memory as the untrusted server. In this particular setting, ORAM can be considered as a structured encryption scheme. The client can run a search program on the server to retrieve the desired data so that the server does not know what type of data or what file has been retrieved. When compared to traditional SSE, ORAM offers better security guarantees if employed *correctly*. For sake of clarity, let us take the following example. First, imagine that a client generates a single keyword SSE construction, outsources the encrypted

[5] We will give further details about STE security definition and leakage functions in Section 2.2.2

[6] Dynamic constructions also leak the update leakage. This leakage is out of scope of this dissertation.

data structure, then generates several number of queries for the same keyword. The server then directly infers that the client is looking for the same keyword even if the entire data is encrypted. In ORAM, this is not the case, the server cannot determine whether the client is looking for similar queries. We say that ORAM hides the search pattern. This is a very desirable property, especially in cases where reducing information disclosure is highly appreciated in the community. There are many other hindrances of applying ORAM to a search context that we will detail later in Section 2.2.3. Leaking less comes at a considerable cost. ORAM main metric is the communication overhead. In general ORAM schemes are based on memory shuffling techniques. These operations are expensive. In the following, for sake of clarity, we focus on three main challenges: communication overhead, storage overhead and security. There are many other challenges to overcome but they are inherent to the instantiation itself rather than the ORAM conceptual design.

2.1.2.1 Communication overhead

The number of bits transferred per every access is the main and most important metric for ORAM schemes comparison. In most ORAM constructions, the data is stored in the form of data blocks. The block is the smallest communication unit employed to compare between ORAM constructions. For instance, the work by Goldreich and Ostrovsky [70] shows that in order to retrieve a block of data, then there is a need to communicate a polylogarithmic number of blocks between the server and the client (we refer here to the hierarchical solution). Authors have also shown that there exists a lower bound on ORAM constructions which is exactly logarithmic in the number of blocks, i.e., the best ORAM construction can at most achieve a logarithmic multiplicative overhead. On the other hand, the communication overhead falls within two types: an amortized and a worst case communication overhead. At a higher level, the amortized setting denotes the average number of blocks that the client communicates to the server after a polynomial number of requests. Most of prior ORAM works have a heavy shuffling step, i.e., after a specific period of time the client has to perform a heavy computational/communication work that consists of downloading the entire data from the server and refreshing it. This represents the worst case communication overhead of an ORAM. Clearly, this is an overwhelming step for the client and represents the main reason that

separated ORAM from a real life deployment for many years. Recently, researchers [129] have shown how to get the shuffling costs down to be polylogarithmic instead of linear in the number of blocks. This can be further decreased when relaxing ORAM model from a storage server to a computational one, as we are going to see in my PhD contribution where the eviction becomes for free [108]. Another dimension to look at was also to relax the single server setting to multiple servers. In this case, we show that ORAM can decrease the communication lower bound while inducing a very small computation overhead mainly based on XOR operations.

2.1.2.2 Storage overhead

Storage overhead represents the second main metric of comparison for ORAM constructions. For all ORAM constructions, the storage consists of two types of blocks: real and dummy blocks. A real block is the meaningful data to be stored in the server, while a dummy block is a fake block used to mislead the server. The storage overhead challenge therefore consists of decreasing the number of dummy blocks in the construction.

2.1.2.3 Security

While ORAM is a primitive that was introduced to prevent software reverse-engineering, re-purposing this primitive comes at the cost of misunderstanding its security when applied to other settings, and to search scenarios in particular. Obliviousness does not hold in search application due to many reasons: in traditional ORAM security definitions, we require that an access stands for retrieving one block. Whereas in search setting, it is very unlikely to fetch the same number of blocks throughout the entire search process unless enforced by a padding mechanism. Second, one can use ORAM as the underlying STE construction so the client can interact with this structure to look up documents' identifiers matching a particular query. However, the final goal of the client is eventually to download the matching documents. That is, the client will send (in the second round) pointers to fetch the files from the main memory/hard drive. The server can easily correlate these two accesses and find out which documents have been fetched. In this case, ORAM does not hide the access pattern. To sum up, the main point is that ORAM does not grant *automatically*

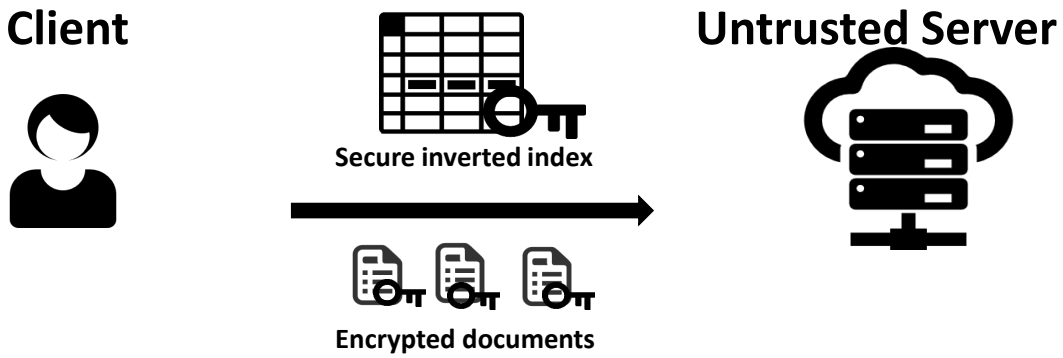

Figure 2.1: Setup phase: the user outsources all files in an encrypted form along with the secure searchable index.

obliviousness except if the entire data is stored in an ORAM, padded to a particular length and every query is upper bounded by a publicly known threshold set at the setup phase[7].

2.2 Scheme Definitions

2.2.1 Setting definition

We start first by describing the common setting for searching over encrypted data. We consider a user $\mathbf{U}$ who wants to store a set of n documents $\mathcal{D} = (D_1, \cdots, D_n)$ to an untrusted server $\mathcal{S}$. Every document consists of a set of keywords subsets of the universal dictionary $\mathrm{W} = (w_1, \cdots, w_l)$. The goal of the user $\mathbf{U}$ is to outsource documents $\mathcal{D}$ to the untrusted server $\mathcal{S}$ while preserving her confidentiality as well as search capabilities. The process of data outsourcing is divided in two phases: a *setup* phase and a *query* phase. We picture the process in Figure 2.1 and 2.2. We give more concrete details in the following section.

2.2.2 Structured Encryption (STE)

Symmetric searchable encryption can be generalized to a broader primitive class, referred to as structured encryption [42] (STE). A STE scheme encrypts data structures in such a way that they can be privately queried. There are several natural forms of structured encryption. The

[7]These patches discussed so far to use ORAM as a sub-component for searching over encrypted data apply only to exact keyword search. It is not clear that using ORAM to handle more expressive queries such as *boolean* queries can provide obliviousness even when applying the above countermeasures.

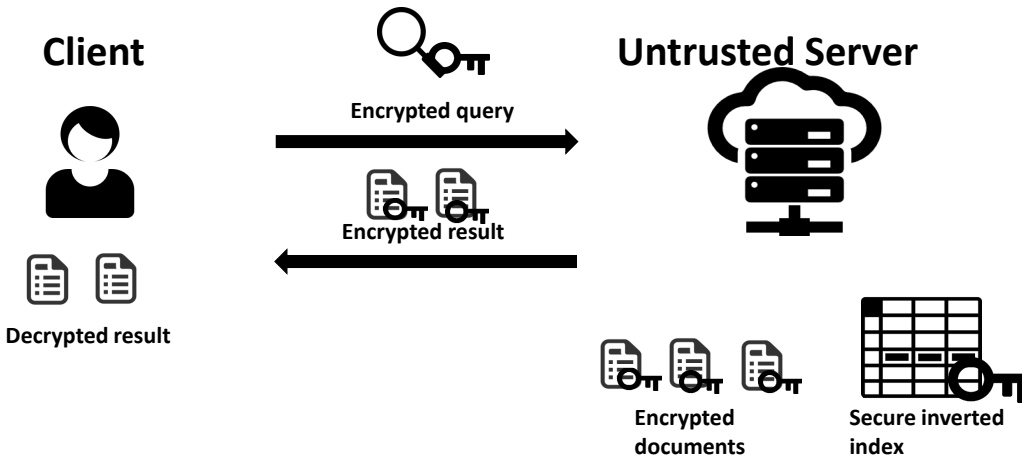

Figure 2.2: Query phase: (1) user sends an encrypted query that the server processes, (2) the server outputs the encrypted files without any knowledge of the query or files plaintext, and (3) the user decrypts the files.

original definition of [42] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [43], the authors also describe *structure-only* schemes which only encrypt structures. Another distinction can be made between *interactive* and *non-interactive* schemes. Interactive schemes produce encrypted structures that are queried through an interactive two-party protocol, whereas non-interactive schemes produce structures that can be queried by sending a single message, i.e, the token. One can also distinguish between *response-hiding* and *response-revealing* schemes: the former reveal the query response to the server whereas the latter do not. Response-revealing denotes the fact that the server can retrieve the matching pointers from the encrypted structure and directly fetch the encrypted files from memory without any interaction with the user.

At a high-level, non-interactive STE works as follows. During a setup phase, the client constructs an encrypted structure EDS under a key K from a plaintext structure DS. The client then sends EDS to the server. During the query phase, the client constructs and sends a token tk generated from its query q and secret key K. The server then uses the token tk to query EDS and recover either a response r or an encryption ct of r depending on whether the scheme is response-revealing or response-hiding.

Definition 2.2.1 (Response-revealing structured encryption [42])**.** *A response-revealing structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ *consists of three polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*: is a probabilistic algorithm that takes as input a security parameter* 1^k *and a structure* DS *and outputs a secret key* K *and an encrypted structure* EDS*.*
- $\mathsf{tk} \leftarrow \mathsf{Token}(K, q)$*: is a (possibly) probabilistic algorithm that takes as input a secret key* k *and a query* q *and returns a token* tk*.*
- $\{\perp, r\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a deterministic algorithm that takes as input an encrypted structure* EDS *and a token* tk *and outputs either* $\perp$ *or a response.*

We say that a response-revealing structured encryption scheme Σ *is correct if for all* $k \in \mathbb{N}$*, for all* $\mathsf{poly}(k)$*-size structures* $\mathsf{DS} : \mathbf{Q} \rightarrow \mathbf{R}$[8]*, for all* (K, EDS) *output by* $\mathsf{Setup}(1^k, \mathsf{DS})$ *and all sequences of* $m = \mathsf{poly}(k)$ *queries* $q_1, \ldots, q_m$*, for all tokens* tk_i *output by* $\mathsf{Token}(k, q_i)$*,* $\mathsf{Query}(\mathsf{EDS}, \mathsf{tk}_i)$ *returns* $\mathsf{DS}(q_i)$ *with all but negligible probability.*

Definition 2.2.2 (Response-hiding structured encryption [42])**.** *A response-hiding structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ *consists of four polynomial-time algorithms such that* Setup *and* Token *are as in Definition 2.2.1 and* Query *and* Dec *are defined as follows:*

- $\{\perp, \mathsf{ct}\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a deterministic algorithm that takes as input an encrypted structured* EDS *and a token* tk *and outputs either* $\perp$ *or a ciphertext* ct*.*
- $r \leftarrow \mathsf{Dec}(K, \mathsf{ct})$*: is a deterministic algorithm that takes as input a secret key* k *and a ciphertext* ct *and outputs a response* r*.*

We say that a response-hiding structured encryption scheme Σ *is correct if for all* $K \in \mathbb{N}$*, for all* $\mathsf{poly}(k)$*-size structures* $\mathsf{DS} : \mathbf{Q} \rightarrow \mathbf{R}$*, for all* (K, EDS) *output by* $\mathsf{Setup}(1^k, \mathsf{DS})$ *and all sequences of* $m = \mathsf{poly}(k)$ *queries* $q_1, \ldots, q_m$*, for all tokens* tk_i *output by* $\mathsf{Token}(K, q_i)$*,* $\mathsf{Dec}_K(\mathsf{Query}(\mathsf{EDS}, \mathsf{tk}_i))$ *returns* $\mathsf{DS}(q_i)$ *with all but negligible probability.*

[8]For some data structure DS, we write $\mathsf{DS} : \mathbf{Q} \rightarrow \mathbf{R}$ to mean that DS has query and response spaces $\mathbf{Q}$ and $\mathbf{R}$, respectively.

2.2.3 Tree-based Oblivious RAM

ORAM allows for an access pattern to an adversarially controlled RAM to be effectively obfuscated. Conceptually, a client's data is stored in an encrypted and shuffled form in the ORAM, such that accessing pieces of data will not produce any recognizable pattern to an adversary which observes these accesses. Similar to RAM accesses, the client performs *Read* and *Write* operations in an oblivious manner such that the server cannot distinguish between both of these operations. At a high-level, the server's memory is shuffled after every access made by the client. The shuffling is often based on a client / server interaction that refreshes some memory blocks, re-encrypts and changes the blocks' order.

In the following, we particularly focus on tree-based ORAM being the most efficient ORAM schemes with poly-logarithmic worst case in the number of blocks. In particular, we will be interested in describing tree-based ORAM's semantics [61, 65, 129, 135]: *Add*, *ReadAndRemove*, and *Evict* operations. We refer readers interested in formal definitions of RAM (being an interactive Turing machine between the CPU and memory) to [70].

At a higher level, tree-based ORAM organizes the outsourced memory in the form of a κ-ary tree (often κ set to 2). Every node in the tree is composed of memory blocks. These buckets are composed of *real* and *dummy* blocks. We will see later that the size of the buckets is a security (or statistical) parameter. For a given address a and a data block d, to simulate ORAM $\mathit{Read}(a)$ and $\mathit{Write}(a, d)$, the client performs a $\mathit{ReadAndRemove}(a)$ followed by $\mathit{Add}(a, d)$. For the correctness of tree ORAM schemes, the client has to invoke an *Evict* operation after every *Add* operation. Let us assume that every leaf has a unique identifier called *tag*. Every block stored is uniquely defined by its address a. We denote by $\mathcal{P}(t)$ the path (the sequence of nodes) containing the set of buckets starting from the root to a leaf identified by its tag t. If $\mathcal{P}(t)$ and $\mathcal{P}(t')$ represent two paths in the ORAM tree, the least common ancestor, $LCA(t, t')$, is uniquely defined as the deepest bucket in the intersection $\mathcal{P}(t) \bigcap \mathcal{P}(t')$. Moreover, to keep track of real blocks' distribution in the ORAM tree, the client stores a lookup table that associates the block identifier to its tag in a *recursive map* (called also position map). In this dissertation, we use the terms node and bucket interchangeably.

Each bucket comprises a set of z entries. We explain *Add*, *ReadAndRemove*, and *Evict* operations in the following.

- *Add*(a, d): To add data d at address a, the client first downloads and decrypts the bucket ORAM of the root of the tree from the server. The client then chooses a uniformly random tag t for a. The tag t uniquely identifies a leaf where d will percolate to. The client writes d and t in an empty entry of the bucket, encrypts the whole bucket, and uploads the result to the root bucket. Finally, the recursive map is updated, i.e., the address a is mapped to t.

- *ReadAndRemove*(a): To read an element at address a, the client fetches its tag t from the recursive map which identify a unique leaf in the tree. The client then downloads and decrypts the path $\mathcal{P}(t)$. This algorithm outputs d, the data associated to a, or $\perp$ if the element is not found.

There are two different ORAM categories. The first one is a "memoryless setting", where the client has constant size (in N) memory available. The second one, "with memory", assumes that the client has a local memory storage that is poly-log in N. For each category, we use different eviction techniques that we present in the following two paragraphs.

Constant Client Memory: The eviction operation is directly performed *after* an *Add* operation. Let us denote by t the leaf tag and by χ the eviction rate.

Evict(χ, t): For each level of the tree, the client chooses from all nodes that are on the same level, respectively, random subsets of $\chi \in \mathbb{N}$ nodes. For every chosen node, the client randomly selects a single block and evicts it to one of its children. The client write dummy elements to all other children to stay oblivious.

Poly-Log Client Memory: For the case of poly-log client memory, the eviction operation follows that of [65, 135]:

Evict(t): Let $\mathcal{P}(t)$ denote the path from the root R to the leaf with tag t. Every element of a node in $\mathcal{P}(t)$ is defined by its data and unique tag t'. For eviction, the client pushes every element in the nodes in the path $\mathcal{P}(t)$, which are tagged with leaf t', to the bucket $LCA(t, t')$.

The eviction operation is performed at the *same* time as an *Add* operation. Instead of storing the element in the root bucket during the *Add* operation, the client performs an *Evict*. Thus, they store, and at the same time evict, all elements as far as possible "down" on the path. Eviction can be deterministic [65] or randomized [135].

Deterministic vs Randomized eviction. Eviction is the most important ceonceptual step in tree-based ORAM as it is the one responsible of shuffling the memory. There are many parameters to take into account during an eviction step such that: (1) making sure that buckets (nodes of the tree) will not overflow, i.e., the buckets will not contain more real elements than their fixed size, (2) the eviction should be as cheap as possible otherwise it annihilates any communication saving won in *Read* and *Write* operations. Deterministic eviction refers to a deterministic reverse lexicographic eviction where the paths of the tree are evicted in an a-priori fixed order. Researchers [65] have shown that deterministic eviction is much simpler with a much clearer overflow proof. Randomized eviction refers to the fact that the evicted path is chosen randomly based on the block being read. Since the blocks' tags are randomly assigned, the evicted paths are then also randomly chosen. While this eviction insures ORAM correctness, the overflow analysis is much complicated when compared to the deterministic one.

2.3 Security Definitions

Security definition, at a higher level, can be seen as a global framework that defines what an adversary cannot break in a given construction based on some a-priori assumptions.

Every security definition takes into consideration two primordial aspects: the *break* and the *adversarial power*. The break describes information that should not be leaked to the adversary (attacker). For example, for private encryption schemes such as AES (Advanced Encryption Standard), we aim to hide the content of any message encrypted by a given key, and even stronger, we aim to prevent the adversary from being able to compute any function based on the ciphertext (encrypted text). The second aspect deals with the computational power of the adversary as well as the conditions under which the attack was performed. The computational power of any adver-

sary is assumed to be within a polynomial-time framework, while the conditions of the attack can be different from one scheme to another depending on the level of security we aim to provide. There are some constructions that are secure even under the assumptions of unbounded adversarial power. This is the case of information-theoretic constructions. Well established security definitions already exist for classical cryptographic primitives such as pseudo-random permutation, pseudo-random function, secret key cryptosystems. The protocols we describe such as STE and ORAM are based on such primitives that we detail later in Section 2.4.

Searching over encrypted data brought some new threats to the user's confidentiality that were not included in the adversary model before. It has been shown that even encrypting the data is not enough to protect its contents from a determined adversary who can still learn some information just based on the search interaction. This situation can get worse since these extra-information along with some auxiliary knowledge can allow the attacker to recover the plaintext data [36, 78, 113, 142]. Researchers [50] in this field noticed this problem and introduced the so-called *leakage functions* in SSE that capture extra-information learned by the adversary (eventually the server) during the protocol execution. We assume in our studies a *honest-but-curious* server, i.e, the server will perform *correctly* the protocol but can gather any extra information outputted during the internal states of the protocol execution.

Leakage can be divided into two types: a setup leakage $\mathcal{L}_{\mathsf{S}}$ and a query leakage $\mathcal{L}_{\mathsf{Q}}$. Note that the leakage description is different from a scheme to another. Here, we will be interested to giving a high level description of information leaked in a single keyword SSE [50]. In the case of conjunctive, substring, boolean SSEs, the leakage often increases (to preserve efficiency). On the other hand, ORAM construction hides some part of the query leakage as will be explained in Section 2.3.2. Below, we go over the details of setup and query leakage.

Setup Leakage $\mathcal{L}_{\mathsf{S}}$ Setup leakage is the information inferred by the server from the encrypted structures. That is, the server can infer some information related to the outsourced data before performing any search operations. The setup leakage (also called static leakage) consists of the size of stored data, the number of unique keywords in the entire data or the maximum number of

files associated to a specific keyword. While these information seem harmless, in some settings, it might be sensitive. As an instance, a client who wants to hide the number of transactions per month can be obliged to pad the data-structure.

Query Leakage $\mathcal{L}_{\mathsf{Q}}$ With respect to the encrypted search problem, *search pattern* and *access pattern* represent the main query leakage. Informally, the search pattern leakage denotes the knowledge captured by the adversary about the queries' occurrence. If the queries are deterministically encrypted, then the same queries have exactly the same ciphertext. Thus, the server infers that these encrypted queries were generated for the same keyword (query). Access pattern leakage embodies a transcript of all the associations between the query and the matching data. In other words, the access pattern leakage captures the accessed memory for every search operation. If a user accesses twice the same memory blocks to retrieve the encrypted documents during two different search operations, the server will find out that the issued query was exactly the same even if the queries were randomized. It is clear that search and access patterns are both correlated, dismissing any of them seems totally useless and can be very misleading, see [112].

The query leakage aims to characterize an adversary who can learn more information during the execution of the protocol. While it is really highly desirable (more secure) to hide any defined leakage function, achieving it will be more complicated and challenging in terms of scheme's design, in particular, in terms of the introduced search/storage overheads. Consequently, many techniques, including STE and even ORAM, do not hide totally or partially from the server these leakage functions. The setup and query leakage can lead, especially in the case of PPE based constructions, to content / query disclosures [36, 78, 113, 142]. In the following, we formally present search and access pattern leakages' definitions for the case of single SSE scheme:

Definition 2.3.1. ***Search pattern leakage****: Given a set of queries $\mathbf{Q} = (q_1, \cdots, q_s)$, the search pattern over $\mathbf{Q}$ is a binary symmetric square matrix Q of size s^2 where $\mathsf{Q}_{i,j} = 1$ iff $q_i = q_j$ and 0 otherwise.*

Definition 2.3.2. ***Access pattern leakage****: Given a set of queries* $\mathbf{Q} = (q_1, \cdots, q_s)$*, the access pattern over* $\mathbf{Q}$ *is the one-to-one association set defined as* $\mathcal{D}(\mathbf{Q}) = ((q_1, \mathcal{D}(q_1)) \cdots (q_s, \mathcal{D}(q_s)))$ *where* $\mathcal{D}(q_i)$ *is the set of documents' identifiers matching the query* q_i*.*

Note that there are some information that cannot be efficiently hidden from the server such as the size of communication interaction with the server, the time and period of protocol execution. These meta-information can be critical in some scenarios but will not be taken into account in our paper and are therefore out of scope.

2.3.1 STE security definition

The standard notion of security for structured encryption guarantees that an encrypted structure reveals no information about its underlying structure beyond the setup leakage $\mathcal{L}_{\mathsf{S}}$ and that the query algorithm reveals no information about the structure and the queries beyond the query leakage $\mathcal{L}_{\mathsf{Q}}$. If this holds for non-adaptively chosen operations then this is referred to as non-adaptive semantic security. If, on the other hand, the operations are chosen adaptively, this leads to the stronger notion of adaptive semantic security. This notion of security was introduced by Curtmola *et al.* in the context of SSE [50] and later generalized to structured encryption in [42].

Definition 2.3.3 (Adaptive semantic security [42, 50])**.** *Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ *be a response-revealing structured encryption scheme and consider the following probabilistic experiments where* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator,* $\mathcal{L}_{\mathsf{S}}$ *and* $\mathcal{L}_{\mathsf{Q}}$ *are leakage profiles and* $z \in \{0,1\}^*$*:*

- $\mathbf{Real}_{\Sigma,\mathcal{A}}(k)$*: given* z *the adversary* $\mathcal{A}$ *outputs a structure* DS*. It receives* EDS *from the challenger, where* $(K, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*. The adversary then* adaptively *chooses a polynomial number of queries* $q_1, \ldots, q_m$*. For all* $i \in [m]$*, the adversary receives* $\mathsf{tk} \leftarrow \mathsf{Token}(K, q_i)$*. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

- $\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k)$*: given* z *the adversary* $\mathcal{A}$ *generates a structure* DS *which it sends to the challenger. Given* z *and leakage* $\mathcal{L}_{\mathsf{S}}(\mathsf{DS})$ *from the challenger, the simulator* $\mathcal{S}$ *returns*

an encrypted data structure EDS *to* $\mathcal{A}$*. The adversary then* adaptively *chooses a polynomial number of operations* $q_1, \ldots, q_m$*. For all* $i \in [m]$*, the simulator receives a tuple* $\big(\mathsf{DS}(q_i), \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i)\big)$ *and returns a token* tk_i *to* $\mathcal{A}$*. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

We say that Σ *is adaptively* $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}})$*-semantically secure if for all* PPT *adversaries* $\mathcal{A}$*, there exists a* PPT *simulator* $\mathcal{S}$ *such that for all* $z \in \{0,1\}^*$*, the following expression is negligible in* k*:*

$$|\Pr[\,\mathbf{Real}_{\Sigma,\mathcal{A}}(k) = 1\,] - \Pr[\,\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k) = 1\,]|$$

The security definition for *response-hiding* schemes can be derived from Definition 2.3.3 by giving the simulator $\big(\perp, \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i)\big)$ instead of $\big(\mathsf{DS}(q_i), \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i)\big)$.

2.3.2 ORAM security definition

Any ORAM construction should meet the typical obliviousness requirement, stated below.

Definition 2.3.4. *Let* $\overrightarrow{a} = \{(op_1, d_1, a_1), (op_2, d_2, a_2), \ldots, (op_M, d_M, a_M)\}$ *be a sequence of* M *accesses* (op_i, d_i, a_i)*, where* op_i *denotes a* ReadAndRemove *or an* Add *operation,* a_i *the address of the block, and* d_i *the data to be written if* $op_i =$ Add *and* $d_i = \perp$ *if* $op_i =$ ReadAndRemove*.*

Let $A(\overrightarrow{a})$ *be the access pattern induced by sequence* $\overrightarrow{a}$*,* k *is a security parameter, and* $\mathsf{negl}(k)$ *a negligible function in* k*. We say that an ORAM construction is secure* iff*, for any PPT adversary* $\mathcal{D}$ *and any two same-length sequences* $\overrightarrow{a}$ *and* $\overrightarrow{b}$*, access patterns* $A(\overrightarrow{a})$ *and* $A(\overrightarrow{b})$*,* $|\Pr[\mathcal{D}(A(\overrightarrow{a})) = 1] - \Pr[\mathcal{D}(A(\overrightarrow{b})) = 1]| \leq \mathsf{negl}(k)$.

Important remark. As pointed out earlier, ORAM primitive can be considered as an STE primitive when applied as a solution to searching over encrypted data problem. It is sometime called *leakage-free* STE construction, refer to [81]. In this case, ORAM primitive can hide the search pattern but not the access pattern in general. In the case of tree-based ORAM construction, the block size needs to be equal to the maximum number of documents associated to every keyword in order to hide the query leakage. Moreover, the documents have to be stored as well in an ORAM to hide the access pattern as the search pattern can be recovered by the length and frequency of documents'

lengths. This composition's fallacy has been discussed by Naveed in his paper, see [112]. Hiding both the search and access pattern can be extremely inefficient and has been dismissed from real life deployments. While ORAM is considered as the holly grail for some problems, it has some leakage when applied to searching over encrypted data.

2.4 Cryptographic Primitives

In this section, we recall some useful and classical definitions in cryptography, we refer the reader to Katz and Lindel book for more details [87].

Definition 2.4.1. ***Negligible function:*** *We say that a function f is negligible in k if $\forall \epsilon > 0$, there exists n_ϵ such for all $k \geq n_\epsilon$,*

$$f(k) \leq \frac{1}{\mathsf{poly}(k)},$$

where $\mathsf{poly}(k) = k^c$, for $c > 0$. We denote a negligible function in k by $\mathsf{negl}(k)$

Definition 2.4.2. ***Advantage:*** *Let $\mathcal{A} : \{0,1\}^n \to \{0,1\}$ be an efficient probabilistic polynomial time algorithm, and R_1, R_2 two random variables. We define the **distinguishing probability** of $\mathcal{A}$, or the **advantage** of $\mathcal{A}$, for R_1 and R_2 as the ability to distinguish between these two random variables:*

$$\mathsf{Adv}(\ A) = |\Pr[\mathcal{A}(R_1) = 1] - \Pr[\mathcal{A}(R_2) = 1]|$$

Definition 2.4.3. ***Symmetric encryption:*** *a symmetric encryption scheme SKE consists of three algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. The key generation Gen takes a security parameter k as input and returns the secret key K, we write $K \leftarrow \mathsf{Gen}(1^k)$. The encryption algorithm Enc takes the secret key, the plain text $m \in \{0,1\}^*$ and a random string r as inputs and returns a ciphertext $\mathsf{ct} \in \{0,1\}^*$, we write $\mathsf{ct} \leftarrow \mathsf{Enc}_K(m, r)$, sometimes we omit the random string such that $\mathsf{ct} \leftarrow \mathsf{Enc}_K(m)$. The decryption algorithm Dec is deterministic and takes the secret key and the ciphertext as inputs and returns the plaintext m or $\perp$ (in case of error) and we write respectively : $\{m, \perp\} := \mathsf{Dec}_K(\mathsf{ct})$, the symmetric encryption in the subsequent chapters is chosen to be semantically secure.*

Definition 2.4.4. ***Semantic security[68]:*** *a symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is semantically secure if for every probabilistic polynomial time algorithm $\mathcal{A}$ there exists a prob-*

abilistic polynomial time algorithm $\mathcal{A}'$ such that for every probability ensemble $\{X_k\}_{k\in\mathbb{N}}$ with $|X_k| \leq \mathsf{poly}(k)$ where k is the security parameter; every pair of polynomial bounded functions $f, h : \{0,1\}^* \to \{0,1\}^*$ we have:

$$\Pr[\mathcal{A}(1^k, \mathsf{Enc}_K(X_k), h(1^k, X_k) = f(1^k, X_k)] < \Pr[\mathcal{A}'(1^k, 1^{|X_k|}, h(1^k, X_k) = f(1^k, X_k)] + \mathsf{negl}(k),$$

where $K \leftarrow \mathsf{Gen}(1^k)$

The probability in these terms is taken over X_k as well as over the internal coin tosses of either algorithms $\mathsf{Gen}, \mathsf{Enc}$ and $\mathcal{A}$ or $\mathcal{A}'$. $\{X_k\}_{k\in\mathbb{N}}$ represents the distribution of plaintexts, f represents the information that the adversary tries to obtain and h represents a priori partial information about the plaintext. A semantic secure scheme SKE is also indistinguishable against chosen plaintext attack experiments, and we say in this case that SKE is IND-CPA secure, or CPA secure.

Definition 2.4.5. ***Pseudo-random functions:*** *Let us consider a function $f : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^l$ and the family F of all maps $\{0,1\}^n \to \{0,1\}^l$. We say that f is a pseudo-random function if:*

- *f is efficiently computable in polynomial time in k.*
- *For any probabilistic polynomial algorithm $\mathcal{A}$:*

$$\mathsf{Adv}(\mathcal{A}) = |\Pr[\mathcal{A}^{f(K,.)} = 1] - \Pr[\mathcal{A}^{r(.)} = 1]| < \mathsf{negl}(k)\ ,$$

  with $K \xleftarrow{\$} \{0,1\}^k$ and $r \xleftarrow{\$} F$.

When f is bijective (in this case $n = l$) then it is a pseudo-random permutation*.*

Definition 2.4.6. ***Pseudo-random generators:*** *Let us consider a function $G : \{0,1\}^k \to \{0,1\}^l$ where $l \geq k$, we say that G is a pseudo random generator if:*

- *G is efficiently computable in polynomial time in k,*
- *For any probabilistic polynomial algorithm $\mathcal{A}$ we have :*

$$\mathsf{Adv}(\mathcal{A}) = |\Pr[\mathcal{A}(G(u)) = 1] - \Pr[\mathcal{A}(r) = 1]| < \mathsf{negl}(k)$$

  where $u \xleftarrow{\$} \{0,1\}^k$ and $r \xleftarrow{\$} \{0,1\}^l$.

Chapter 3

Related Works

> Knowledge is a process of piling up facts;
> wisdom lies in their simplification.
>
> *Martin Luther King, Jr.*

Encrypted search can be solved using different cryptographic primitives offering different levels of security, functionality and efficiency. Based on the requirements of the setting along with the client's needs, different choices can be made. Encrypted search can be solved using the following cryptographic primitives:

- symmetric searchable encryption (SSE)
- oblivious RAM (ORAM)
- secure multi-party computation (MPC)
- private information retrieval (PIR) (single or multiple servers)
- full homomorphic encryption (FHE)
- property-preserving encryption (PPE)

In this chapter, we will mainly focus on detailing the state of the art of oblivious RAM (ORAM) and symmetric searchable encryption (SSE). There are two reasons behind our choice: first, SSE and ORAM are the two primitives that provide a reasonable balance between security, efficiency and functionality, second and more importantly, SSE is a cryptographic primitive designed particularly for search contrary to all other listed primitives that were rather introduced for more general reasons.

At a conceptual level, one can consider that both SSE and ORAM can be considered a possible instantiation of structured encryption. Structured encryption (STE), introduced by Chase and Kamara [42], shows how to encrypt any data structure and privately query on it later on. If

ORAM is employed as a component to search on data, it can be considered as a *leakage-free* STE construction[1].

Before detailing the state of the art of both SSE and ORAM, we give in the following a brief overview of PPE, PIR, MPC and FHE.

Property preserving encryption Property-preserving encryption is an encryption technique that preserves a specific property between plaintexts. As an instance, if two plaintext messages are equal, then their ciphertexts will be equal as well. Examples of these properties are: equality, order and orthogonality. Deterministic encryption [28] is an equality-preserving encryption scheme while order preserving encryption schemes [26, 27] preserve the order of the plaintexts.

Formally, given two messages m_1 and m_2 from a message space $\mathcal{M}$, an encryption scheme $\mathsf{DTE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an equality-preserving if for all $K \leftarrow \mathsf{Gen}(1^k)$, we have $\mathsf{Enc}_K(m_1) = \mathsf{Enc}_K(m_2)$ if $m_1 = m_2$.

Similarly, given two messages m_1 and m_2 from a message space $\mathcal{M}$, then an encryption scheme $\mathsf{OPE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is order-preserving if for all $K \leftarrow \mathsf{Gen}(1^k)$, we have $\mathsf{Enc}_K(m_1) \odot \mathsf{Enc}_K(m_2)$ if $m_1 \odot m_2$, where $\odot \in \{<, >, =\}$.

PPE has gained an increasing interest from the database community due to its high efficiency. Particularly, PPE constructions make encrypted search easier to incorporate over existing systems such as in relational database management systems. PPE constructions are legacy-friendly and can be incorporated in a *plug-and-play* manner without any major change of the underlying system design and architecture. This is an extremely appealing feature and it was first explicitly introduced by the work of Popa et al. [122]. This work was influential in that many products were built on the same design such as Cipherbase, SEEED, Microsoft Always Encrypted SQL Servers and Google Big Queries [1, 2, 14, 20].

However, from a security perspective, both deterministic and order preserving encryption schemes fail to provide the needed security insurance for the encrypted search problem. These two prim-

[1]In this statement, we have made several assumptions in order to claim that ORAM is a leakage-free construction that implies a non-trivial padding. Depending on the way the data structure is stored in the ORAM, the leakage can consist at least of the maximum response length. While the search pattern is always randomized in ORAM constructions, the access pattern is not totally hidden as the length of the response can be correlated to the query itself.

itives are considered to be extremely leaky and even deprecated from being used in practice for some scenarios. For a deterministic encryption to be secure, the theoretical security definition requires having a minimum amount of entropy (min-entropy) in the data to be encrypted, but this requirement does not hold in a real-life scenario. The data encrypted by the users are not sampled randomly from a uniform distribution. Recently, Naveed et al. [113] showed that CryptDB-like systems and all property-preserving encryption based encrypted database systems (EDB) in general can leak tremendous amount of information when applied to store electronic medical records (EMR). Recently, a new line of research started to investigate the security guarantees of PPE- based EDBs and the security of all cryptographic primitive when applied to encrypted search in general. Cash et al. [36] show that using PPE techniques as a building block for some SSE constructions can also be disastrous as it is the case for ShadowCrypt system [75]. There are also many follow-up papers that attack PPE constructions (order preserving encryption and order-revealing encryption) such as [39, 58] when applied to the EDB systems.

Private information retrieval Private Information Retrieval (PIR) is a cryptographic primitive that enables the user to issue randomized queries to fetch a data block *obliviously*. If we consider that the data is organized in an array A such that each cell consists of a block of data, PIR offers the client the possibility to issue a query q_i to fetch the ith position such that the server's guess about the position that has been fetched is equal to $\frac{1}{\#A}$. Similarly to ORAM, we say that PIR hides the access pattern. PIR is a more powerful primitive in the sense that it can also be applied to plaintext data, i.e., even if the data A does not belong to the user, the client can still fetch the desired data block while the server does not learn the position that has been fetched. PIR was presented for the first time by the work of Chor et al. [45]. This construction sets out the first family of PIR, called multiple-servers PIR or information-theoretic PIR, where the same data is replicated into several non-colluding servers. This category of PIR offers an information-theoretic security, i.e., even if an adversary is computationally unbounded, it cannot break the construction's obliviousness. However, the server holding the replicated data must not collude at any time. There are different constructions [23, 57] that can handle k-out-of-n of corrupted servers that can share

their queries and states. Moreover, researchers also got interested to improve multiple-servers robustness against Byzantine attacks where some of the servers can fail or behave inconsistently. There is also an interesting research result that shows that IT-PIR is equivalent to locally decodable codes [88]. Kushilivitz et al. [91] introduced a single server PIR, dubbed CPIR for computational PIR. This class of PIR considers only a single server in which data (clear or encrypted) is stored. In CPIR, the user makes use of some cryptographic primitives, such as additive homomorphic encryption, in order to retrieve data. While there have been many interesting results that improve the communication complexity of PIR constructions, the main downside of PIR construction (either computational or information-theoretic) is that the computational cost for one search is linear in the entire size of the dataset, i.e., in $O(\#A)$. This is clearly inefficient for search purposes where the search can be performed in optimal time.

Multi-Party computation Multi-party computation (MPC) [25, 52, 53, 54, 69, 93, 94, 100] is one of the most powerful and general cryptographic primitives that can compute a function f securely. In MPC, there are n entities where each has its private input (set of inputs) hidden from other entities. These entities aim to compute a public function f over the set of their inputs. As an instance, f can be a function that computes the average value of the inputs. At the end of the protocol execution, the result of the MPC will be the output of the function f, while no party discloses its own input. Applied to the encrypted search, the MPC parties play the roles of server(s) and/or user(s) with the private inputs respectively equal to the keywords (documents) and the query (range, fuzzy, approximate, distance...). The public function f can be modeled as a boolean or arithmetic circuit. The arithmetic circuit is based on additive and multiplicative gates while the boolean one is based on and, or, and negation gates. MPC can be considered as the most secure cryptographic primitives. In fact, while the server in most SSE and ORAM constructions is considered as an honest-but-curious entity, MPC provides a general framework where an entity may actually cheat and falsify the result of the protocol. For example, the server will not send or even tamper with the correct matching values. MPC, under some assumptions, can enable the parties to detect such behavior. From a security perspective, MPC hides the search and access

pattern, i.e. the server will not discern which document matches which query (hidden access pattern), and the query will always be randomized (hidden search pattern). In term of efficiency, MPC is extremely inefficient as it has to take the entire database as input. This is inherent to the circuit representation of the search function f as the parties have to feed the entire data to the circuit to evaluate the function. Recently, Gordon et al. [74] show how to combine ORAM and secure MPC to reduce the computation overhead to be sub-linear instead of being linear.

Full homomorphic encryption Full homomorphic encryption (FHE) [64] is a public-key encryption scheme that can support arbitrary computation. Given a set of encrypted message $\text{ct}_1 = \mathsf{Enc}(K_p, m_1), \cdots, \text{ct}_m = \mathsf{Enc}(k_p, m_n)$, the evaluator (eventually the server) evaluates any arbitrary function f such that $\mathsf{Eval}(\text{ct}_1, \cdots, \text{ct}_m) = \mathsf{Enc}(k_p, f(m_1, \cdots, m_n))$. FHE can be also used to search on encrypted data as follows: the client encrypts its dataset $\mathsf{DB} = (m_1, \cdots, m_n)$ and sends it to the server. The client wants to verify whether a keyword m exists in the dataset DB. The client sends $\mathsf{Enc}(K_p, m)$ along with a multi-variate public function f that outputs one if $m \in \mathsf{DB}$ and zero otherwise. The server will then evaluate the function and sends back the encrypted results that the client decrypts locally. FHE is very expressive but also very expensive as its search time is linear with an expensive evaluation cost.

To sum up, we briefly detailed PPE, PIR, MPC and FHE as tools that can be used to search over encrypted data, however, they have different limitations when it comes to solve the encrypted search problem. We have shown above that PPE has serious leakages issues, while MPC, FHE, and PIR are too inefficient to be employed to search over encrypted data. Nevertheless, we want to emphasize that our assumptions do not dismiss the fact that some of these solutions can be eventually used as sub-components for more efficient constructions as it is the case of using CPIR in our C-ORAM [108] construction detailed in Chapter 5. We also employ multi-servers PIR in our Oblivious P2P construction along with a seed homomorphic pseudo-random generator, we refer the reader to our paper [80] for more details.

3.1 Searchable Encryption

The area of searchable encryption has been an active research area for over a decade. Many original constructions for searchable encryption can be found in the literature both in the symmetric search setting [38, 41, 50, 66, 130] as well as the asymmetric one [16, 24, 29, 49]. Symmetric searchable encryption was first explicitly solved by Song et al. [130]. Symmetric schemes, in general, are more geared towards cloud storage and archival for a single user. Asymmetric schemes were first introduced by the work of Boneh et al. [29]. Asymmetric schemes are more suitable for multiple users in a collaborative environment. Early schemes focused only on exact keyword searches. In the asymmetric setting, several schemes have been proposed with support for conjunction, range or subset searches [30, 77, 119, 127, 128, 137] that exploit the mathematical properties (in particular, homomorphism or pairing) inherent in this setting. However, asymmetric schemes remain orders of magnitudes less efficient than symmetric ones and are not suitable for practical deployments. Moreover, asymmetric SSE constructions have some security limitations as well that make them not suitable for data outsourcing. For example, the work by Boneh et al. [29] demonstrates how a client can search on emails encrypted by other users using the client's public key. The client issues a trapdoor to search for a particular keyword. This trapdoor will be later evaluated against all encrypted keywords and will output either true or false depending on whether the email contains the keyword or not. This same trapdoor can be stored by the server for future use. As the server has the client's public key, it can then encrypt many keywords against which it can evaluate all recorded trapdoors. This brute force attack can determine exactly the plaintext that the client searched for.

Extending single keyword SSE Symmetric searchable encryption has been generalized to structured encryption (STE) by Chase and Kamara [42]. In [42], the authors show how to encrypt different data structures such as graphs, matrices and multi-maps. Since then, SSE has been enhanced to handle dynamic settings where the user can add, delete and update files on the encrypted structure stored on the server side [37, 85, 86, 114, 131]. In the symmetric setting, Curtmola et al. showed that it was possible to support multiple users [50]. Recent works have also studied several exten-

sions of symmetric searchable encryption schemes to support conjunctions in sub-linear time, such as, [38] (with an extension in the multi-user setting as well presented in [79]). Faber et al. [59] extend the work by Cash et al.[38] and demonstrate how to handle range search, substring, wildcard and phrase search. Cash et al. [37] also provided several constructions that are I/O efficient that can be made dynamic. Cash and Tessaro introduce lower bounds on SSE for a better locality [40]. Recently, Asharov et al. [21] provide SSE constructions that have optimal locality, space overhead with a nearly-optimal search efficiency. Meng et al. [104] show how to encrypt graphs and how to perform approximate shortest distance queries.

Security in SSE Reducing the leakage is extremely important in structured encryption. Zhang et al [142] showed that for some particular scenarios where an adversary can inject forged files in the encrypted data structure, the adversary can learn the encrypted queries generated by the client. They showed that *forward-secrecy* is an important security property that helps avoiding file injection attacks. Forward-secrecy, at a higher level, is a feature that helps the client to disassociate the update queries from the search queries. That is, if the client performs multiple search queries and then runs an update, the server (adversary) will not associate the update to any previous search query. Most of the dynamic SSE constructions are not forward-secure and are vulnerable to injection attacks. On the other hand, file injection attacks have some strong assumptions such that the adversary has to inject a large number of files composed of different keywords. While injection can occur more or less smoothly in an email system with poorly designed spam filtering, these assumptions are far from being realistic in normal outsourcing scenarios where a client outsources only data belonging to him. Recently, Bost [31] presented a forward-secure construction that is much more efficient than the one presented by Shi et al. [131]. We want to emphasize, however, that forward-secrecy only protects (or to be more rigorous delays) the file-injection attacks. That is, for a scenario in which file injection attacks can occur, a forward-secure STE (SSE) is not sufficient to stop the attack.

There have been many attacks recently that analyze the security of SSE constructions. Islam et al. [78] demonstrate that under some knowledge of the user's data set, the adversary can recover

the plaintext queries performed by the client. Cash et al. [36] improve the IKK attacks and show that only leveraging some setup leakage can lead to recover the plaintext data in some PPE-based SSE schemes. Recently, Pouliot and Wright [123] present a new attack on efficiently deployable PPE-based-SSE construction that also lead to recover the content plaintext.

SSE and database community The database community has also looked into this problem with solutions that focus on the performance and practicality at the cost of lower security guarantees. For instance, the SADS system presented by Raykova et al. in [124] and then extended in several aspects in [118], involve two semi-trusted parties that facilitates the search. Leveraging on these semi-trusted entities, the authors propose a document retrieval protocol that supports disjunctive and conjunctive queries (which implies also range queries) and runs in time proportional to the size of the returned set of documents. However, the main weakness with respect to other protocols is the requirement of having two semi-trusted parties that fulfill the index server and query router roles. The project BlindSeer [117] proposes a system that combines rich functionality (including support for arbitrary Boolean formula in a sub-linear time), practical scalability and provable security (leakages are clearly identified and controlled). However, this work has a setting different from the traditional user /server setting with new involved trusted entities non-existing in traditional SSE settings. An SSE scheme, however, can be extracted though from [117] and can be compared to the OXT scheme as well. Finally, Kamara and Moataz [84] present the first encrypted relational database, SPX, based solely on structured encryption. They show that SPX leaks much lesser than previous constructions based on property-preserving encryption.

3.2 Oblivious RAM

Oblivious RAM goes back to the seminal paper by [70]. There have been several attempts to improve different aspects of ORAM, such as its communication complexity, number of interactions between the server and the client, memory complexity on the client side, and storage and computation overhead on the server [46, 55, 65, 67, 70, 71, 72, 73, 90, 101, 105, 108, 109, 116, 121, 125, 129, 135, 138, 139]. We briefly review three ORAM categorizations. The first discusses

recent advances of schemes with constant client memory complexity, the second targets schemes with sublinear client memory, and the third presents recent works in multiple-servers ORAM.

Constant client memory. Constant client memory is very appealing for resource-constrained devices with limited memory, e.g., embedded devices, small sensors, and devices in the Internet of Things. [71] and [121] introduced amortize communication complexity in $O(\log^2 N)$, but with linear worst-case communication complexity. [129] introduce tree-based structures providing a worst-case poly-logarithmic communication complexity in $O(\log^3 N)$ blocks. Many subsequent papers build on top of this one to further decrease communication or storage complexity storage [65, 101, 105, 109]. Recently, there have been many attempts to decrease the communication overhead to be constant in the number of blocks. That is, obliviously reading or writing a block with only a constant number of transferred blocks as overhead. Using servers with computational capabilities instead of storage-only servers, Devadas et al. [56] showed how to construct a constant communication ORAM for blocks in $\Omega(\log^5 N)$. Fetcher et al. [62] show how to decrease the number of interactions of Onion ORAM from $\log N$ to 1. Moataz et al. [108] demonstrate how to preserve constant communication for smaller block size in $\Omega(\log^4 N)$, while performing eviction with fewer number of homomorphic multiplications. Although low asymptotic bounds have been reached for communication complexity, high computational latency on server side makes constant client memory not yet ready for deployment [108].

Poly-log client memory. Earlier schemes have memory complexity on the client side in $O(\sqrt{N})$, yet inducing a linear worst-case communication complexity [138, 139]. Stefanov et al. [134] show how to get a worst-case memory complexity in $O(\sqrt{N})$ with a a communication complexity in $O(\log^2 N)$. Stefanov et al. [135] present how to provide a $O(\log N)$ communication complexity with only a logarithmic memory complexity on the client size. This scheme has been improved by multiplicative constant in [105, 125]. Recently, [63] improves the number of interactions of Path ORAM to be constant while inducing a multiplicative security overhead factor.

Distributed setting. Many ORAMs leverage multiple servers to decrease overhead. For example, ObliviStore [133] decreases overhead using an oblivious load balancing technique relying on *trusted* internal nodes to distribute accesses. Stefanov and Shi [132] constructions accesses blocks in $O(1)$ in a two-servers setting (extendable to k servers) with $O(\log N)$ communication complexity between servers and $O(\sqrt{N})$ client storage complexity. Lu and Ostrovsky [98] show how to achieve Goldreich and Ostrovsky's lower bound $O(\log N)$ with two non-communicating servers and with $O(1)$ client storage complexity. Dachman-Soled et al. [51] introduce the notion of oblivious network RAM that can be also fit to a distributed setting and decreases access complexity to $O(1)$. However this comes at the cost of a weaker security model where the adversary is only allowed to observe communication between servers and client. Servers themselves are trusted to not reveal details about queries.

Chapter 4

Expressiveness in Symmetric Searchable Encryption

You never fail until you stop trying.

Albert Einstein

Chapter Outline. In this chapter, we present our SSE contributions:

1. BSSE, a boolean symmetric searchable encryption in Section 4.1. We present one of the first boolean symmetric searchable encryption schemes in literature[1]. Based on an original idea that transforms keywords to an orthonormal family, we enable secure boolean queries by means of simple and efficient inner product. Our scheme can handle any CNF/DNF formula and has a linear search complexity.

2. 3SE, semantic SSE in Section 4.2. We present the first semantic SSE construction. We start by showing that existing stemming techniques used in plaintext search constructions can be easily employed to partially solve the semantic search over encrypted data. We demonstrate that by leveraging standard optimal time SSE constructions such as [37, 42, 50], we can build a semantic SSE that has the same efficiency and security insurance with better expressiveness.

3. SED, a substring SSE in Section 4.3 (on-going work). In this work, we initiate the study of substring search over encrypted data. We show that by leveraging the same orthogonalization technique employed in the BSSE construction, we can make substring search sub-linear over encrypted data — as long as the size of keywords dictionary is negligible in the number of keywords. We proceed by introducing two gradually enhanced constructions. We have implemented our final construction and show that it demonstrates of its readiness for a real life deployment.

[1] Cash et al. [38] show the same year, 2013, how to handle sub-linear conjunctive queries over encrypted data.

4.1 BSSE: Boolean SSE

4.1.1 Contribution Summary

BSSE was the first structured encryption scheme to support arbitrary boolean queries over encrypted data. It is based on an original idea of considering keywords as vectors and using the Gram-Schmidt process to orthogonalize and then orthonormalize them. It further makes use of a very efficient operation, the inner product, to perform searches at the server side. The inner product indeed leverages the orthonormalized keywords to efficiently test if a boolean expression query matches the label corresponding to an encrypted document or not. It is an application of mathematical and computing principles to practical security, especially in the searchable encryption scope.

In addition to that, our BSSE tries to introduce a new conceptual approach, proper to oblivious RAM and PIR to the SSE setting. BSSE's query sent for retrieving encrypted documents is *randomized*, which means that the server sees different queries even if the same boolean query is sent several times. On the other hand, the label that we associate with each encrypted document can be seen as the secure index defined by Goh [66] or the encrypted data structure defined by Curtmola et al. [50]. However, the secure index leaks some meta-information about the number of keywords stored in the Bloom filter, while our secure labels do not *apriori* leak such information about its content or about the number of keywords that are encrypted before being sent to the server. Our contribution is based on an adversarial model similar to the one defined by Curtmola et al. [50], the difference only arises in the nature of the query sent to the server. However, one can find out that the BSSE encrypted data structure can leak more than the size of the dataset. One can show that, without performing any queries, the server can recover the number of common keywords between two documents while leveraging the orthogonality feature. The same attack can be leveraged to find some relations between the queries. Nevertheless, this boils down to consider the orthogonalization *only* as a process to speed up the search at the expense of larger storage while improving the scheme expressiveness. We will give more details about the security of orthogonalization in Section 4.3.

4.1.2 BSSE construction

Our overall approach is to consider the keywords as vectors and to pad them so that they form a family of independent vectors. From such a family we construct an orthonormal basis of the resulting vector space, which means that each keyword is associated with one vector of this orthonormal base. Then, to each document we associate the vector resulting of the addition of all vectors. The queries are then also expressed as a vector in the same vector space using the same basis and we roughly use scalar product between the vector of the query and the vector of a document to decide whether a document satisfies a query or not. We thus first present how to construct the orthonormal basis and then present the full detail of our construction.

4.1.2.1 BSSE pre-construction

The BSSE algorithm is based on a pre-construction over the unique keywords set $\mathrm{W} = \{w_1, ..., w_l\}$. The aim of this construction is to create from W an orthonormal keyword family. This last is obtained by applying the Gram-Schmidt algorithm to a free family derived from W.

To be more precise, we consider all keywords to be of equal length m (this can be achieved by padding all keywords so that they have the same length).

Let consider a pseudo-random permutation $\pi : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m$ and a key K_1 generated from the $\mathsf{Gen}(1^k)$ where k is a security parameter, for each keyword w_i in W, we apply π_{K_1} in order to create a permuted set of keywords such that the new set of keywords $\mathrm{W}_\pi = \{\pi_{K_1}(w_1), ..., \pi_{K_1}(w_l)\}$.

Let us consider the canonical base of size l, $\{e_1, ..., e_l\}$ such that $\forall i \in \{1, l\}\ |e_i| = l$, and $e_i = (\delta_{i,1}, \ldots, \delta_{i,j}, \ldots, \delta_{i,l})$.[2] In order to create a free family $(f_i)_{i\in[l]}$ from W_π, we concatenate every permuted keyword $\pi_{K_1}(w_i)$ with e_i such that $f_i = \pi_{K_1}(w_i)\|e_i$ and $|f_i| = m + l$.

We consider the inner scalar product ϕ :

$$\begin{array}{rccc} \phi & : & \{0,1\}^{m+l} \times \{0,1\}^{m+l} & \rightarrow & [m+l] \\ & & x, y & \mapsto & \sum_{k=1}^{m+l} x_k y_k \end{array}$$

Theorem 4.1.1. *If $(f_i)_{i\in[l]}$ is a free family of a pre-Hilbert space, there exists an orthogonal family $(o_i)_{i\in[l]}$ such that:*

[2] $\delta_{i,j}$ is called the Dirac function such that $\delta_{i,j}$= 1 if $i = j$ and 0 otherwise.

- $\phi(o_i, o_j) = \phi(o_i, o_i)\delta_{i,j}$,
- $\mathsf{Vect}(f_1, ..., f_l) = \mathsf{Vect}(o_1, ..., o_l)$, *where* Vect *represents the space generator.*
- *The scalar product* $\phi(f_i, o_i)$ *is strictly positive*,

Definition 4.1.1. *The orthogonal projection* p *on a vector line led by* u *is defined as:*

$$p_u(v) = \frac{\phi(v, u)}{\phi(u, u)}u.$$

Orthogonal Keyword Construction: Let us consider $(f_i)_{i\in[l]}$ a free family. Then, there is an orthogonal family $(o_i)_{i\in[l]}$. We can construct this family following the Gram-Schmidt process:

$$o_i = f_i - \sum_{s=1}^{i-1} p_{o_s}(f_i), \text{ for } i \in [l].$$

$(o_i)_{i\in[l]}$ forms the new keyword orthogonal family that verifies:

$$\forall i, j \in [l],\ \phi(o_i, o_j) = \phi(o_i, o_i) \cdot\ \delta_{i,j}. \tag{1}$$

In addition, we can normalize the vectors to obtain an orthonormal family:

$$\forall i \in [l],\ u_i = \frac{1}{\sqrt{\phi(o_i, o_i)}} \cdot o_i.$$

At the end of this phase, we should point out that each orthonormal keyword u_i corresponds to a keyword $w_i \in \mathrm{W}_o$. In addition, if we take into account the order of the free family $(f_i)_{i\in[l]}$, then the construction according to the Gram Schmidt algorithm produces a unique orthonormal family. If we change the order of keywords, the algorithm outputs a different orthonormal family. Note that the number of possible families generated by this construction is $l!$.

We should also emphasize that the keyword set W will be stored on the client side in a table $\mathcal{T}$, and that optionally this table could also contain the orthonormal keywords corresponding to them W_o . Here ends the preliminary setup phase, in the following, we will detail BSSE algorithms, namely, the setup and search steps.

In the following, we capture the entire pre-construction process in a map functionality Ortho that takes a set of strings W and a key K and outputs an orthonormal set of strings W_o. We denote by $\mathsf{Ortho}(K, w)$ the orthonormal keyword corresponding to w.

4.1.2.2 Construction of BSSE algorithms

BSSE is a tuple of three polynomial time algorithms such that $\mathsf{BSSE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$. In the following, we provide details of each of the algorithms of BSSE.

$\mathsf{Setup}(1^k, \mathcal{D})$. Considering a semantically secure encryption $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, the Setup algorithm takes as inputs the security parameter k, a collection of documents $\mathcal{D}$, and the set of unique keywords W an it works as follows:

- generate two secret keys $K = (K_1, K_2)$ such that $K_1 \xleftarrow{\$} \{0,1\}^k$ and $K_2 \leftarrow \mathsf{Gen}(1^k)$.
- based on the entire set of unique keywords $\mathrm{W} = \{w_1, \cdots, w_l\}$, produce the orthonormal keyword family $\mathrm{W}_o = \{u_1, \cdots, u_l\} \leftarrow \mathsf{Ortho}(K_1, \mathrm{W})$. These orthonormal keywords can either be retrieved or recomputed depending on whether they were stored in table $\mathcal{T}$ or not.
- for every document $D_i \in \mathcal{D}$ that corresponds to a set M_i of unique orthonormal keywords $(u_j)_{j \in M_i}$, output the encrypted document $\mathsf{Enc}_{K_2}(D_i)$, and the associated label $\mathcal{X}_i = \sum_{j \in M_i} u_j$ for each $i \in [n]$. We denote by $\mathsf{EDB} = ((\mathcal{X}_i, \mathsf{Enc}_{K_2}(D_i))_{i \in [n]})$.

Once all documents are encrypted, they will be stored in the outsourced server with their associated labels, see table 4.1.

$\mathsf{Enc}_{K_2}(D_1)$	$\mathcal{X}_1$
...	...
$\mathsf{Enc}_{K_2}(D_i)$	$\mathcal{X}_i$
...	...
$\mathsf{Enc}_{K_2}(D_n)$	$\mathcal{X}_n$

Table 4.1: Encrypted documents along with encrypted indexes in the outsourced server

For both Token and Query algorithms, for clarity, we consider presenting three incremental constructions. We first show how to proceed with conjunctive queries, then how to take into account disjunctions, and finally show to deal with negations and hence any boolean expression in disjunctive normal form (DNF).

Conjunctive Case. Let us consider the conjunctive query $\mathbf{w} = w_1$ AND w_{i_2}...AND w_r. The generation of the token works as follows:

$\mathsf{Token}(K, \mathbf{w})$**.**

- convert the keywords in $\mathbf{w}$ to the corresponding orthonormal keywords either by re-generating this base by re-performing the pre-construction, or by retrieving the corresponding vectors from table $\mathcal{T}$ such that $\mathbf{w} = \mathsf{Ortho}(K_1, w_1)$ AND $\cdots$ AND $\mathsf{Ortho}(K_1, w_r)$.
- pick uniformly at random r strictly positive integers $\{a_i\}_{i\in[r]}$ such that $a_i \xleftarrow{\$} \{0,1\}^k$. Output

$$\mathsf{tk} = \frac{1}{\sum_{i\in I_q} a_i} \cdot \sum_{i\in I_q} a_i \cdot \mathsf{Ortho}(K_1, w_1),$$

where I_q denotes the indices of keywords in W.

$\mathsf{Query}(\mathsf{tk}, \mathsf{EDB})$**.** Parse $\mathsf{EDB} = ((\mathcal{X}_i, \mathsf{Enc}_{K_2}(D_i))_{i\in[n]})$, for all $i \in [n]$, the server will perform the following verification:

$$\begin{aligned}
\phi(\mathsf{tk}, \mathcal{X}_i) &= \mathsf{tk} \cdot \mathcal{X}_i^\top \\
&= \frac{1}{\sum_{j\in I_q} a_j} \sum_{j\in I_q} \sum_{l\in M_i} a_j \cdot \phi(u_j, u_l) \\
&= \frac{1}{\sum_{j\in I_q} a_j} \sum_{j\in I_q \bigcap M_i} a_j,
\end{aligned}$$

where I_q represents the set of keywords' indices that are being searched for.

- If all keywords exist in the ith document, i.e., $I_q \bigcap M_i = I_q$ then: $\phi(\mathsf{tk}, \mathcal{X}_i) = 1$. Send $\mathsf{Enc}_{K_2}(D_i)$ to the user.
- Else, the ith document does not contain all keywords, i.e., $I_q \bigcap M_i \neq I_q$ and then : $\phi(\mathsf{tk}, \mathcal{X}_i) \neq 1$.

Conjunctive and disjunctive search: In this paragraph, we consider a more general case where the user wants to search for the following query composed of t disjunctions: $\mathbf{w} = \bigvee_{z=1}^{t} \bigwedge_{j\in I_{q_z}} w_j$, where I_{q_z} represents the set of all keywords' indicies in the query $\mathbf{w}$.

$\mathsf{Token}(K, \mathbf{w})$**.** The user translates the keywords to the corresponding orthonormal keywords. For each $z \in [t]$, the user picks at random $|I_{q_z}|$ integers $\{a_{z,i}\}_{i \in |I_{q_z}|}$ such that $a_{z,i} \xleftarrow{\$} \{0,1\}^k$. We then define:

$$\mathcal{S}_z = \frac{1}{\sum_{j \in I_{q_z}} a_{z,j}} \sum_{j \in I_{q_z}} a_{z,j} u_j$$

Output the token tk such that:

$$\mathsf{tk} = \begin{pmatrix} \mathcal{S}_1 \\ \vdots \\ \mathcal{S}_t \end{pmatrix}$$

$\mathsf{Query}(\mathsf{tk}, \mathsf{EDB})$**.** Parse $\mathsf{EDB} = ((\mathcal{X}_i, \mathsf{Enc}_{K_2}(D_i))_{i \in [n]})$, for all $i \in [n]$, the server will perform the following verification:

$$\begin{aligned} \phi(\mathsf{tk}, \mathcal{X}_i) &= \mathsf{tk} \cdot \mathcal{X}_i^\top = \begin{pmatrix} \phi(\mathcal{S}_1, \mathcal{X}_i) \\ \vdots \\ \phi(\mathcal{S}_t, \mathcal{X}_i) \end{pmatrix} \\ &= \begin{pmatrix} \frac{1}{\sum_{j \in I_{q_1}} a_{1,j}} \sum_{j \in I_{q_1} \bigcap M_i} a_{1,j} \\ \vdots \\ \frac{1}{\sum_{j \in I_{q_t}} a_{t,j}} \sum_{j \in I_{q_t} \bigcap M_i} a_{t,j} \end{pmatrix} \end{aligned}$$

The result of this operation is interpreted as follows:

- If $\exists z \in [t]$ such that $(\phi(\mathsf{tk}, \mathcal{X}_i))_z = 1$ then the ith document matches the query, as it means that at least one conjunction was satisfied, hence the disjunction of these conjunctions is satisfied.
- Else if $\forall z \in [t],\ (\phi(\mathsf{tk}, \mathcal{X}_i))_z \neq 1$ then the ith document does not match the query, because no conjunction was satisfied so the whole disjunction is not satisfied either.

So far, we have seen general queries dealing with any number of conjunctions or disjunctions in the same query. However, to complete our scheme, we need to take into account the negation as well. The negation refers to the fact that the client does not desire fetching documents containing a particular keyword. In the following paragraph, we detail our full solution which was the first solution that supports general boolean searches over encrypted data.

Boolean search: We now consider the case of a general boolean expression search taken in the disjunctive normal form as follows: $\mathbf{w} = \bigvee_{z=1}^{t} \bigwedge_{j \in I_{q_z}} \diamond w_j$, where $\diamond \in \{\perp, \neg\}$.

For this case, we need to introduce a new *nosiy* keyword ξ which we assume that it belongs to all documents $\mathcal{D}$. To this extent, we apply the pre-construction Ortho to $\mathrm{W} \cup \{\xi\}$, to output all orthonormal keywords. After the Gram-Schmidt process, we thus have one orthonormal keyword $\mathsf{Ortho}(K_1, \xi)$ that has been added up to all $\mathcal{X}_i$, for all $i \in [n]$.

$\mathsf{Token}(K, \mathbf{w})$**.** Let us now consider every conjunction $L_z = \bigwedge_{j \in I_{q_z}} \diamond w_j$ separately. I_{q_z} represents the indexes of keywords in each disjunction L_z. We split I_{q_z} into two partitions such that: $I_{q_z} = \{P_{q_z}, N_{q_z}\}$ where P_{q_z} and N_{q_z} represent respectively the indexes of affirmative and negative keywords in the boolean expression. For each $z \in [t]$, we pick at random $|P_{q_z}|$ integers $\{a_{z,i}\}_{i \in |P_{q_z}|}$, $|N_{q_z}|$ integers $\{c_{z,i}\}_{i \in |N_{q_z}|}$ and one random integer b_z such that $a_{z,i}, c_{z,i}, b_z \xleftarrow{\$} \{0,1\}^k$. We then define for all $z \in [t]$:

$$\mathcal{S}_z = \frac{1}{b_z + \sum_{j \in P_{q_z}} a_{z,j}} \Big(\sum_{j \in N_{qz}} -c_{z,j} u_j + \sum_{j \in P_{q_z}} a_{z,j} u_j + b_z J \Big)$$

The user sends to the server the following token:

$$\mathsf{tk} = \begin{pmatrix} \mathcal{S}_1 \\ \vdots \\ \mathcal{S}_t \end{pmatrix}$$

$\mathsf{Query}(\mathsf{tk}, \mathsf{EDB})$**.** Parse $\mathsf{EDB} = ((\mathcal{X}_i, \mathsf{Enc}_{K_2}(D_i))_{i \in [n]})$, for all $i \in [n]$, the server will perform the following verification:

$$\begin{aligned} \phi(\mathsf{tk}, \mathcal{X}_i) &= \mathsf{tk} \cdot \mathcal{X}_i^{\top} = \begin{pmatrix} \phi(\mathcal{S}_1, \mathcal{X}_i) \\ \vdots \\ \phi(\mathcal{S}_t, \mathcal{X}_i) \end{pmatrix} \\ &= \begin{pmatrix} \frac{1}{b_1 + \sum_{j \in P_{q_1}} a_{1,j}} \big(\sum_{j \in N_{q_1} \bigcap M_i} c_{1,j} + \sum_{j \in P_{q_1} \bigcap M_i} a_{1,j} + b_1 \big) \\ \vdots \\ \frac{1}{b_t + \sum_{j \in P_{q_t}} a_{t,j}} \big(\sum_{j \in N_{q_t} \bigcap M_i} c_{t,j} + \sum_{j \in P_{q_t} \bigcap M_i} a_{t,j} + b_t \big) \end{pmatrix} \end{aligned}$$

The result of this operation is interpreted as follows:

- If $\exists z \in [t]$ such that $(\phi(\mathsf{tk}, \mathcal{X}_i))_z = 1$ then the ith document exactly matches the query. That is, it means that in the corresponding disjunctions all affirmative keywords are in the document while none of the non negative keywords is present (otherwise the result would be smaller than one).

- Otherwise if $\forall z \in [t]$, $(\phi(\mathsf{tk}, \mathcal{X}_i))_k \neq 1$ then the ith document doesn't match the query.

The server sends back to the user the set of all encrypted documents that match the query.

Note: The reason why we have included a noisy keyword ξ in all documents is for the scheme's correctness. For instance, consider the case where we have only negative keywords in the disjunction L_z. Then $|P_{q_z}| = 0$. Without a joker, the query $\mathcal{S}_z$ would have to be divided by 0, for $z \in [t]$. By adding the noisy keyword, we prevent this case from occurring and the division is defined.

4.1.3 Security Analysis

As discussed in Section 2.3, we need to capture the setup leakage and query leakage of our BSSE construction that we detail below:

$$\mathcal{L}_{\mathsf{S}}^{\mathsf{bsse}}(\mathcal{D}) = \big(n, \{M_i \bigcap M_j, \forall i, j \in [n]\}\big),$$

where M_i is a set containing the evaluation of a random function $f : \{0,1\}^{\log l} \to \{0,1\}^k$ on the keywords in $\mathcal{D}$.

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{bsse}}(\mathcal{D}, \mathbf{w}) = \left(\big(\mathrm{SP}(\mathcal{D}, w_i)\big)_{\substack{i \in I_{q_z} \\ z \in [t]}}, \mathrm{AP}(\mathcal{D}, \mathbf{w})\right),$$

where $\mathbf{w} = \bigvee_{z=1}^{t} \bigwedge_{j \in I_{q_z}} \diamond w_j$, SP and AP denotes the search pattern and access pattern as defined in Section 2.3 respectively.

Theorem 4.1.2. *If* SKE *is semantically secure encryption scheme,* π *a secure pseudo-random permutation, then* BSSE *is* $(\mathcal{L}_{\mathsf{S}}^{\mathsf{bsse}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{bsse}})$*-semantically secure.*

The proof of this theorem is similar to the one that we will be giving for the SED protocol in Section 4.3.

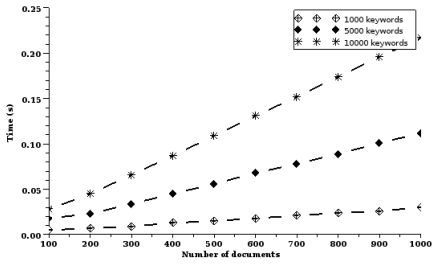

Figure 4.1: Search phase

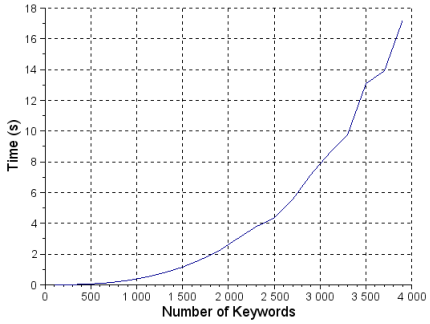

Figure 4.2: Orthogonal keywords construction

Note that contrary to optimal single SSE construction in which we normally only leak the size of the dataset $\mathcal{D}$, in BSSE we also leak the number of common keywords between any two documents in the corpus. This is a consequence of using orthogonality feature to handle better expressiveness. Cash et al. [38] intruduce OXT, a construction that leaks much lesser while achieving better asymptotic. BSSE only advantage is that it can handle arbitrary boolean queries in a linear time. Kamara and Moataz showed later that one can handle arbitrary boolean query with much lesser leakage in a worst-case sub-linear time [82].

4.1.4 BSSE Performance

From a performance perspective, the most important feature to benchmark of any structured encryption is its search phase as it is performed by the server for potentially many users. The search complexity of our construction is $O(n)$ inner products where n is the number of documents stored in the server. This is as good as the solution provided by Goh in [66], except that our scheme can handle much more complex queries. The search phase is illustrated in figure 4.1, in this example, we show that the search is linear in the number of documents, we take also three different cases where documents stored in the server are associated to labels containing 1000, 2000 or 4000 keywords (for all searches we have taken the same boolean query, we did not take into account the network transmission delay or the query construction time). We emphasize that the increase of labels size implies a longer computation phase for every document.

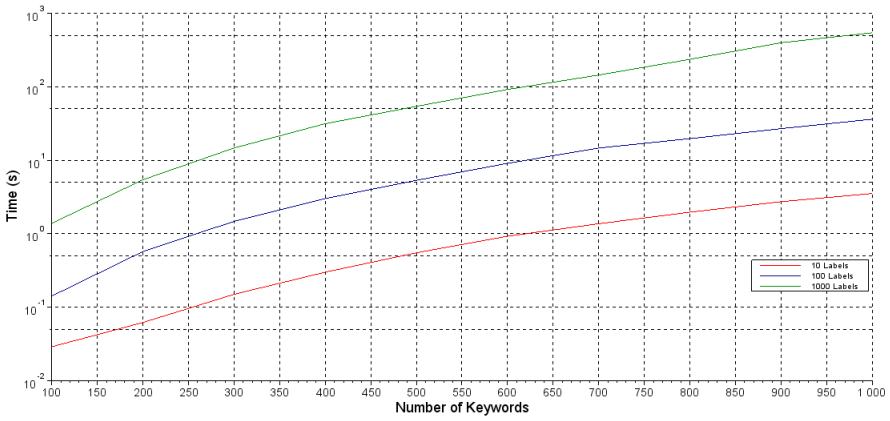

Figure 4.3: Labels creation

The main novelty in our scheme is the pre-computation phase with the Gram-Schmidt process. As we mentioned there are two options: either performing this computation once and storing the result in a table resulting in $O(l)$ storage complexity or trading storage for computation and performing the Gram-Schmidt process for each query. Intuitively, since storage is cheap the first solution would be more advantageous. However we implemented the keyword orthonormalization construction which include the Gram-Schmidt and the permutation processes and found out that this computation phase is still reasonable although quadratic in the number of keywords as can be seen in Figure 4.2.

In this figure, the *number of keywords* denotes the size of the free family that we have constructed in the first section. As can be seen on the graph, the construction of one thousand orthogonal keywords takes less 0.5 second which is acceptable given that these computations are performed on a client side. The algorithm used was not optimized in any way, and it simply proves that the solution of recomputing the base at each time is feasible (although we believe that the storage alternative is more efficient). In addition to this implementation, we have tested the performance of the off-line construction of labels in the client side, see Figure 4.3 (we do not take into account the extraction of keywords from documents).

Section Conclusion. We introduced in this section a boolean structured encryption scheme. The search overhead is linear which makes BSSE interesting only for small data sets as has been shown in the evaluation section. Whereas we have described the leakage profile of BSSE, we believe that this leakage can be harmful in many scenarios and practitioners need to be aware of it.

On the other hand, we want to point out that BSSE was introduced concurrently to many other boolean constructions [38, 117]. The scheme by Pappas et al. [117] outperforms BSSE in all dimensions while the Cash et al. [38]'s construction does better in the conjunctive case. We have recently introduced the first STE construction that can perform arbitrary boolean search operations in a sub-linear time with optimal communication while being non-interactive. Our solution can be extended to be dynamic and forward-secure. We do not detail it but we refer the reader to [82] for more details about our construction.

4.2 3SE: Semantic SSE

4.2.1 Contribution Summary

In this section, we present 3SE an original construction that enables semantic search over encrypted data. To achieve this goal, we first studied the state of the art techniques to perform semantic search over plaintext data, as such techniques enhance the performance and more generally the efficiency of information retrieval mechanisms. Most of these techniques are based on a variety of stemming algorithms. This technique has shown its efficiency for plaintext data, thus we aim to use it in an encrypted context.

From this point, our contribution is to combine an efficient searchable encryption technique with a stemming algorithm in order to perform semantic search over encrypted data. In the basic searchable encryption scheme, one keyword is associated to all of its corresponding documents. In order to allow semantic search, we have to associate all related keywords to the same set of documents. Related keywords have the same root and consequently we can only store the root of related keywords in the encrypted data structure. This will greatly reduce the storage overhead as we are going to detail later on. Throughout the search phase, the user selects the keyword that he wants to search for, then automatically, the stemming algorithm extracts the root of this keyword

and finally the root is sent instead of the keyword in the query. As a result, the user retrieves all encrypted documents which contain all related keywords. By mixing one of the most efficient searchable encryption schemes [37, 42, 50, 82] and a stemming algorithm we can thus achieve semantic searches over encrypted data. The stemming process, however, needs to be fixed during the setup phase and the client cannot change it as it will miss with the scheme's correctness. In the following, we briefly give a high level overview of the existing stemming algorithms that can be used as a component of 3SE.

4.2.2 Stemming Algorithms

In this section, we present stemming algorithms which are fundamental building blocks of most semantic search mechanisms. For this purpose we present on the one hand an overview of stemming algorithms and depict features that they achieve, and on the other hand we give a brief but exhaustive classification of these algorithms.

4.2.2.1 Stemming Algorithms Overview

Stemming is a general morphological process analyzer which aims first at identifying words having roughly the same meaning, in particular words which have the same root, and associates this set of words with the root (called also stem if no prefixes were removed from this set of words during the process). We should point out that despite the fact that all stemming algorithms rely on morphological language aspects, the purpose of stemming algorithm is not to find the correct meaningful root of words. Indeed, the root might very well have an incorrect form with respect to the lexical rules, for example the word "ease" can have "eas" as a root result. On the contrary, the process aiming at normalizing words and finding the correct morphological stem is called the lemmatization: the latter proceeds with algorithms which find the canonical form of lexemes (lexemes represent keywords which have the same root). In semantic search algorithms, the query stemming is performed obliviously for the user, thus the morphological incorrectness of word's stems is not a problem, and having "ease" or "eas" will not change anything for the query results. Another point to emphasize is that basic stemming algorithms do not take into account

the grammatical or lexical relationship, instead they assume that words having the same root are semantically close.

4.2.2.2 Stemming Algorithms Classification

The literature is abundant of stemming algorithms, which take advantage of different techniques such as suffix-stripping algorithms, n-gram algorithms, stochastic algorithms and so on. These algorithms can be classified in three main types: affix stripping, statistical stemming and mixed stemming.

Affix Stripping. Affix stripping algorithms apply procedurally several rules in order to remove known suffixes and prefixes from words in order to identify the "root" . As leading stripping stemmer examples we can quote the first removing stemmer algorithm introduced by J.B. Lovins [97] in 1968 and the well known and most used Porter suffix stemmer algorithm introduced in 1980 [126] and developed into a widespread framework called "Snowball". However this type of algorithm has the disadvantage of imperatively requiring an *a priori* knowledge of the language to build rules of affixes removing. In the following we enter into more details of the Porter algorithm for a better understanding of the affix stripping algorithms.

Porter algorithm. Porter algorithm aims at removing suffixes from words in order to find the root. We can imagine the following set of terms : CONNECT, CONNECTED, CONNECTING, CONNECTION, CONNECTIONS (example taken from [126]); these words are lexemes which have similar meaning, and the stem of these lexemes is CONNECT. The operation of finding the stem is done by automatic means: the algorithm defines a set of rules (about 60 but they can be extended further) and words are then confronted to all these rules sequentially before outputting the stem. The rules have the following form: (condition)(old suffix)$\Rightarrow$(new suffix). Here are for instance two examples of rules:

if the word length > 0 *and ends in 'ing', remove the 'ing'.*

if the word has at least one consonant and one vowel and ends in 'eed', replace the 'eed' by 'ee'.

The latter means for example that “agreed” becomes “agree” while “feed” remains unchanged. In summary Porter’s algorithm is quite simple and its wide spread implementation “Snowball” reflects its popularity. The algorithm is not generic in that one needs to define different rules for different languages, but once these rules are defined for a language the algorithm performs well and can have many applications.

Statistical Stemming. As explained in the previous subsection, the knowledge of the language used is the major disadvantage in stemmer algorithms. To overcome this limitation, Mayfield and McNamee introduced the n-gram stemming algorithm [102] which does not require the definition of the language as input. Indeed their idea consists of doing statistical tests on n-grams (recall that an n-gram is a group of n consecutive letters in a word) existing in the document (whatever the document language): the n-grams which have the bigger frequency are considered as affixes; on the contrary the n-grams which have the lowest frequency are classified as roots. This algorithm has good results for stemming without having the knowledge of the language and the authors demonstrated its efficiency by testing it for eight languages. There exists other interesting statistical stemmers besides Mayfield and McNamee’s n-gram method such as the one introduced by Melucci and Orio based on the Hidden Markov Models [103] and the one introduced by Majumder et al. [99] based on clustering words and defining stems as the centroids of each cluster. As the name of this category indicates it, all these stemmers have in common to be based solely on statistics on words or subwords in a text, their performance is good when the language is unknown, but comparatively their performance is lower than affix stripping algorithms if the language is know.

Mixed Stemming. Mixed algorithms aim at overcoming the limitations of the previous approaches namely the context sensitivity and the corpus related features. Indeed, affix stemmers tend to conflate words with similar syntaxes but different semantics. This difference between words having the same root is due to the corpus based issue. On the other hand, we can find several inflected words used in different documents which deal with different contexts and have the same syntactic root, however their semantic meaning is different, an issue that we can tackle by using statistical stemming. Consequently the user ends up retrieving documents which are not

semantically related to its query. Few advanced schemes were proposed to deal with this issue by taking into account the context of words in documents, and we can in particular cite the corpus-based approach [141] by Xu and Bruce and the more recent development described by Peng et al. in [120].

This concludes the overview and classification of stemming algorithms which are used in particular to perform semantic searches over plaintext data. In the next section we shift our focus to one of the most efficient searchable encryption mechanism [50][3].

4.2.3 Semantic SSE: Construction Overview

As stated in the introduction, our aim is to enable semantic search over encrypted data. Our approach is to combine stemming algorithms with searchable encryption ones.

4.2.3.1 Constructions

Naive Approach. The first idea is to simply combine stemming and searchable encryption sequentially without modifying any of them. Given an STE construction such that $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$, we can construct a naive 3SE construction by a simple modification of the Token algorithm. That is, in order to retrieve documents containing keywords having the same meaning as a given keyword, the user could generate from this original keyword the set all the keywords within proximity in a semantic sense. For example, from the keyword "search", the user generates "searching", "searches", "research", etc. Then the user has to send as many queries as the number of semantically close keywords to the server, which answers with the documents corresponding to each keyword. Overall, given a stemming algorithm Stem and a standard single keyword SSE construction $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$, the only modification can be depicted as follows:

$\mathsf{Token}(K, w)$:

- compute the stems $(s_1, \cdots, s_l) \leftarrow \mathsf{Stem}(w)$;
- for all $i \in [l]$, compute $\mathsf{tk}_i \leftarrow \mathsf{STE}.\mathsf{Token}(K, s_i)$;

[3]There are other constructions [37, 38, 42] that have better storage overhead, or slightly better leakage functions. However, all these constructions have the same search efficiency.

- output $\mathsf{tk} = (\mathsf{tk}_1, \cdots, \mathsf{tk}_l)$.

This basic solution satisfies basic requirements of semantic searchable encryption however it has several shortcomings: it requires higher communication overhead and several search computation in the server side (the increase is proportional to the number of semantically close keywords). Furthermore, several semantically close keywords could belong to the same document which will then be sent several times (in the case of a response hiding STE construction, otherwise if the STE is response revealing, then the server performs the union of all documents identifiers before sending them back to the user), which is not optimal. Finally this approach might also be less secure as the server learns that several semantically close keywords are being searched, which is an additional information leakage which impact is not well understood yet. All these drawbacks lead us to consider the following other approach.

Improved Construction. Our main construction is based on a modification of standard single keyword SSE integrating a stemming algorithm Stem throughout the entire process of the protocol 3SE. First of all, we stress that our construction works with any stemming algorithm, however, the choice of the stemming algorithm will implicitly impact the strategy of information retrieval. As shown before, there are several types of stemming algorithms, each one has some features which make it more suited for semantic or syntactic retrieval, and the user has to chose the one which is more appropriate for his strategy. In our first implementations we have decided to use Porter's algorithm [126]. Our construction aims at associating documents with the root of all the related lexemes. This association extends the scope of the encrypted documents retrieval to all keywords sharing the same stem and consequently with nearly the same meaning. We detail our 3SE algorithm in Figure 4.4.

Settings. We give in the following some properties that capture the relation between the keywords and the stems. Given a stemming algorithm Stem which takes as input keywords within W and outputs the stems of all these keywords that we store in the set $\mathcal{S}_t = \{s_1, ..., s_r\}$ such that $|\mathcal{S}_t| < |\mathrm{W}|$. As shown in Figure 4.5, each stem s_i in $\mathcal{S}_t$ is associated to a unique subset of keywords

Let consider $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ be a single keyword SSE construction. Let Stem be a deterministic stemming algorithm. We define $\mathsf{3SE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$, the semantic search over encrypted data algorithm, as follows:

$\underline{\mathsf{Setup}(1^k, \mathcal{D})}$:

- initialize an empty set $\mathcal{S}_t$;
- for all $w \in \mathrm{W}$, generate $s \leftarrow \mathsf{Stem}(w)$, add s to $\mathcal{S}_t$;
- for every stem $s \in \mathcal{S}_t$, associate all documents identifiers in $\mathcal{D}$ in DB_s;
- compute $(K, \mathsf{EDB}) \leftarrow \mathsf{STE.Setup}(1^k, \mathsf{DB}_s)$.

$\underline{\mathsf{Token}(K, w)}$:

- compute $s \leftarrow \mathsf{Stem}(w)$;
- generate $\mathsf{tk} \leftarrow \mathsf{STE.Token}(K, s)$;
- output tk.

$\underline{\mathsf{Query}(\mathsf{tk}, \mathsf{EDB})}$: output $r \leftarrow \mathsf{STE.Query}(\mathsf{tk}, \mathsf{EDB})$.

Figure 4.4: Semantic encrypted search 3SE.

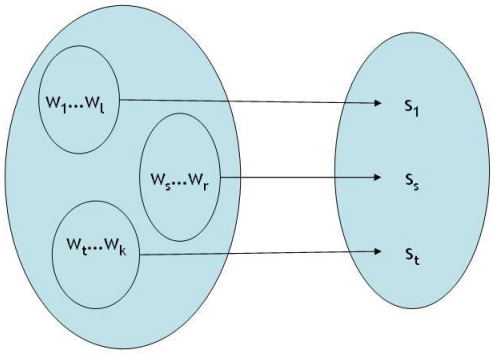

Figure 4.5: Stemming process

denoted by $\mathrm{W}(s_i) \subset \mathrm{W}$. Moreover, since each keyword has one and only one stem, we then have

$$\forall i, j \in [m] \; s.t. \; i \neq j, \; \mathrm{W}(s_i) \cap \mathrm{W}(s_j) = \emptyset.$$

We denote by $\mathsf{DB}_s(s_i)$ the set of all documents' identifiers containing at least one element within the set $\mathrm{W}(s_i)$:

$$\mathsf{DB}_s(s_i) = \bigcup_{w_j \in \mathrm{W}(s_i)} \mathsf{DB}_s(w_j).$$

4.2.3.2 Analysis

The improved construction enables the user to search for all documents containing the keyword w_i and also all the lexemes that are similar to w_i. Similarity is underlined by the fact that the lexemes and the keyword w_i have the same stem. As mentioned previously, retrieving all these documents will enhance the efficiency of the information retrieval for the user as it returns documents which are relevant even though they do not contain the exact keyword. From a functionality perspective, the construction is consistent and achieves the basic features of semantic symmetric searchable encryption.

Security Analysis. From a security perspective, semantic symmetric searchable encryption 3SE is secure in the same sense as the adaptive security by Curtmola et al.'s in [50]. The additional stemming phase is indeed performed locally at client side and does not affect the security of our construction. However the access pattern defined by the association between the query and the

resulting documents is modified and queries return more documents on average in our construction compared to [50]. However, conceptually, the query leakage of 3SE is similar to the one by [50]. We can formally define the leakage as follows:

$$\mathcal{L}_{\mathsf{S}}^{\mathsf{se}}(\mathsf{DB}) = \mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}(\mathsf{DB}_s),$$

and

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{se}}(\mathsf{DB}, w) = \mathcal{L}_{\mathsf{Q}}^{\mathsf{ste}}(\mathsf{DB}_s, s),$$

where $s \leftarrow \mathsf{Stem}(w)$.

This leakage helps an attacker learning more relations between the documents, but on the other hand, retrieving more identifiers constitutes a noise to any adversary who wants to extract information from the access/search pattern.

Theorem 4.2.1. *If* STE *is* $(\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{ste}})$*-semantically secure., then* 3SE *is* $(\mathcal{L}_{\mathsf{S}}^{\mathsf{se}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{se}})$*-semantically secure.*

The proof is straightforward and similar to the one in [50].

Computation Complexity. From a computation perspective, the search phase is also optimal as 3SE uses standard optimal single keyword SSE constructions. Since the stem's set is smaller than the keyword set such that $|\mathcal{S}_t| \leq |\mathrm{W}|$, this implies that in practice the encrypted structure has a lower storage overhead. In terms of communication overhead, the server has to send more documents than for the single keyword SSE constructions. While the stem set is smaller than the keyword set, one stem is often associated to more documents' identifiers, which therefore makes the communication overhead for a semantic SSE higher than single keyword SSE. However, this is an inherent property of semantic SSEs as the user wishes to retrieve more documents when compared to the single SSE case.

Section Conclusion. In this section, we propose 3SE the first semantic SSE construction we are aware of. We show that such construction preserves the optimality of single SSE schemes while providing better security expressiveness. We want, however, to point out that 3SE is far

from achieving full-scale semantic expressiveness such as answering queries of the form: the *most significant* documents or *natural language* queries. For this, one needs to perform non-trivial changes to our construction. We believe that incorporating natural language search capabilities over encrypted data sets will be an interesting research direction that we leave as an open problem for future works.

4.3 SED: Substring SSE

4.3.1 Contribution Summary

In this section, we investigate the problem of efficient substring search over encrypted data. The user stores a set of documents $\mathcal{D} = (D_1, \cdots, D_n)$ in the server. Later, the user submits a query involving a substring of one or more keywords in the dictionary W. The server responds by retrieving the set of documents from the corpus that contain the keywords of which the given string is a substring. We develop a simple protocol that implements such substring search over encrypted data *with search complexity linear in the size of the dictionary where* $|\mathrm{W}| \ll n$, *without inducing any significant storage overhead.*

To solve this problem efficiently we re-formulate the problem to ask the following question: *Is there a bivariate function* f *that takes as input an encrypted substring* s *and a set of encrypted keywords in* W*, and outputs a deterministic boolean result that defines the success or failure of the search?* Orthogonalization of vectors appears to provide some solution to the problem. Two vectors of the same subspace x_1 and x_2 are *orthogonal* if their inner product is equal to 0; x_1 and x_2 are orthonormal if they are orthogonal and they are unit vectors. Orthogonalization is the process of determining a set of such orthogonal vectors spanning a specific subspace. We have employed a similar orthogonalization technique to construct a boolean SSE as detailed in Section 4.1.

We provide an intuition behind our approach. Imagine that two vectors x_1 and x_2 represent two letters say from the English alphabet. If x_1 and x_2 represent the same letter then they are not orthogonal and their inner product will *not be equal to zero*; if x_1 and x_2 represent different letters then the inner product will be *equal to zero.* We extend this idea to search for substrings in keywords. Assume that the letters in an alphabet are represented as column vectors such that these

vectors are pairwise orthonormal. A keyword in a dictionary can be considered as a concatenation of letters. The keyword then is a matrix where each column corresponds to a letter of the keyword and two columns are adjacent if the corresponding two letters are adjacent in the keyword. Given the fact that a substring is also a concatenation of letters and can similarly be represented by a matrix, a matrix product between the substring and the keyword can provide an "answer" matrix for the substring membership. (Recall that the matrix product is composed of a number of inner products between rows of the first matrix and columns of the second one in a specific order.) To be precise, the result will be a matrix with ones in all the diagonal components in case of a substring match. Thus, our bivariate function f can be intuitively viewed as a matrix product operation that takes as an input an orthogonal representation of the substring and orthogonal representation of the keyword. For the actual protocol implementation, the function f takes as the second input a matrix representing all keywords in the dictionary in the transformed format. We call this second input *the searchable part* in our protocols. Note that our actual construction is a little bit more complex and we will discuss this in details later.

Briefly, the basic protocol works as follows. The user generates the set of all keywords W from the documents' collection $\mathcal{D}$. The user then generates the searchable part corresponding to these keywords in a manner discussed above. The user also creates an *inverted index* that associates keywords with their respective documents. Finally, the user encrypts the documents and uploads the searchable part, the inverted index and the encrypted documents to the server. Note that the inverted index is encrypted using single-keyword SSE technique. A user query generation is similar to the generation of any column of the searchable part. During the search phase, the server will first output all indexes (keywords) that contain the given substring (query), then uses the inverted index to retrieve the set of matching encrypted documents with a single interaction with the client.

Note that this technique, although appealing, induces many challenges in term of efficiency, correctness and security. First, characters, in ASCII for instance, are not orthogonal to each other in their binary representation. This is because normal ASCII representation uses 8 bits. In order to be orthogonal they should be linearly independent first, which implies that their binary repre-

sentations should be at least equal to 128 bits. Consequently, a transformation is needed to impart the orthogonality feature. This, in turn, brings up certain challenges in terms of the size of the transformed letter, and a decimal representation that involves computational precision. Second, a matrix product for identifying substring membership tends to be computationally inefficient for large dictionaries; inner product computation improves efficiency. To facilitate inner product computation, a compaction of the query (substring searched for) and the searchable part is mandatory which brings up more challenges for protocol correctness. Third, every keyword has its own ordering between letters that should be taken into consideration. In fact, this adjacency is critical for the correctness of the scheme especially if we want to reduce the size of the query as well as the searchable part. Fourth, the search scheme should ideally allow wildcard search similar to one enabled on plaintext search. Last but not least, substring search tends to leak more information than exact keyword search. To understand why, note that in order to perform encrypted search, a user sends (often) a deterministic token for a given substring. As a result, the server acquires the knowledge of all matching substrings, that is, the server learns that the *token* corresponds to a given set of substrings. Over time the server can build a knowledge of the substring frequencies. Assuming an exact knowledge of the dictionary, the server can thereafter associate any token to their exact plaintext values. SED protocol employs orthogonalization process for the sole purpose of fast inner product and leveraging both orthonormal and additive properties, the security of SED nevertheless consists of other cryptographic primitives involved in the construction process of the searchable part.

In the following, we present two progressively refined Substring search over Encrypted Data SED schemes. The first scheme, SED-1, is the initial attempt at using orthogonalization techniques for encrypted substring search. SED-1 helps us to understand the basic principles behind the searching scheme. We discuss that SED-1 is vulnerable to some forms of inference attacks as well as attacks based on the search history. The SED-2 scheme improves upon SED-1 by reducing information leakage based on characters' positions and pattern's construction. The SED-2 scheme is efficient in that the communication complexity is linear in the size of the substring in the search

query sent to the server, and the search complexity is in $O(|\mathrm{W}| + \mathsf{opt})$, where opt is the number of matching documents.

4.3.2 Substring search over Encrypted Data - Pre-construction

4.3.2.1 Pre-construction

Let us consider the set of letters $\mathrm{l} = \{l_1, \ldots, l_t\}$. The first step in the pre-construction is to create an independent family from these letters and then generate the associated orthogonalized vectors. The orthogonalization, then orthonormalisation, are similar to the one introduced for BSSE in Section 4.1. We refer the reader to that section for details about the pre-construction. One difference between SED and BSSE is that instead of creating a free family for the entire dictionary of keywords, for SED, the free family is generated only for the set of letters composing a specific keywords dictionary. It is clear that the size of this family, in the case of SED, is much smaller when compared to the BSSE one.

We now discuss our first scheme – SED-1.

4.3.3 Substring Search Over Encrypted Data #1 – SED-1

4.3.3.1 Setup phase

Let $\mathcal{D} = \{D_1, ..., D_n\}$ be the set of documents, and $\mathrm{W} = \{w_1, ..., w_{|\mathbf{W}|}\}$ be a set of keywords. We assume that this dictionary is ordered, e.g. pseudo-random evaluation on the lexicographic order. For each keyword w_i in the dictionary W, we denote by $\mathsf{DB}(w_i)$ the set that contains the entire list of document identifiers that contain the keyword w_i.

Next, the user creates an encrypted inverted index $\mathcal{T}$ for the inverted index DB using any of the optimal single keyword SSE constructions such as [37, 38, 42, 50]. The plaintext inverted index DB can be seen as a dictionary data structure and it consists of a number of $\langle label, value \rangle$ pairs, one for each keyword in the dictionary. The entity $label$ is a nonnegative integer equal to the position of the keyword in the dictionary. In other words, the ith $label$ in the inverted index is associated with the keyword w_i in $\mathbf{W}$. We would like to emphasize that we do not store any information related to the actual keyword. The $value$ is the concatenation of all elements in the set of document identifiers,

$\mathsf{DB}(w_i)$, containing the keyword w_i. The client finally outputs the encryption of the inverted index $\mathcal{T}$. This encrypted structure can be queried similarly to the existing single keyword SSE, i.e., by constructing tokens, in our case, based on the index position to retrieve documents identifiers. Given the row number, this encrypted inverted index will help the server during the retrieve phase to find the relevant document identifiers in an optimal search time leveraging the search optimality of the existing constructions [37, 38, 42, 50]. Based on the most efficient construction so far by Cash et al. [37], the size of this inverted index is in $O(N)$, where $N = \sum_{w_i \in \mathbf{W}} \mathsf{DB}(w_i)$.

The third step consists of constructing the searchable part $\mathcal{V}$ of the encrypted data structure using which the server identifies which keywords contain the given substring. To begin with, we construct $\mathcal{V}$ as $\mathcal{V} = \{v_1, \ldots, v_m\}$. The value m represents the size of the longest keyword in the dictionary W. Each vector[4] v_i is a row vector of a size equal to |W|. Each vector v_i contains some value corresponding to the ith letter of each keyword in the dictionary. If the jth entry of the inverted index corresponds to a given keyword w_j, the jth component of the vector v_i will correspond to the ith letter of the same keyword. The vector v_i is generated as follows.

Consider the set of all possible letters $\mathbf{l} = \{l_1, ..., l_t\}$ and a special symbol ξ that will be used to denote a 'noisy' position for keywords with a size lower than m. We apply the pre-construction Ortho on $\mathbf{l} \bigcup \{\xi\}$ phase in order to generate the orthonormal set of letters. In the position j of the vector v_i, we insert the orthonormal letter that exists in the ith position of the jth keyword. The vector v_i is then equal to:

$$v_i = (u_{i,1}, u_{i,2}, \ldots, u_{i,|\mathbf{W}|}),$$

where $u_{i,j}$ is a column vector of the orthonormal ith letter of the keyword w_j such that $u_{i,j} = \mathsf{Ortho}(\chi, w_j[i])$. If the size of the keyword w_j is smaller than m, we put the orthonormal value $\mathsf{Ortho}(\chi, \xi)$ in all the corresponding positions of v_i where $|w_j| + 1 \leq i \leq m$.

We reiterate the same process for all m vectors. At the end, we obtain the following vector set $\mathcal{V} = \{v_1, \ldots, v_m\}$ that represents the searchable part associated with all possible positions of letters in keywords. At the end of the setup phase, the user sends to the server the collection $(\mathsf{Enc}_K(\mathcal{D}), \mathcal{T}, \mathcal{V})$, where $\mathsf{Enc}_K(\mathcal{D})$ represents the encryption of the document collection where

[4]The vector v_i is a $(t \times |\mathrm{W}|)$ matrix, but for sake of clarity we will call it a vector.

$K \leftarrow \mathsf{Gen}(1^k)$, $\mathcal{T}$ the encryption of the inverted index based on [37] construction, and $\mathcal{V}$ the searchable part described above.

Storage Overhead. The storage complexity of the SED-1 scheme is $O(|\mathcal{T}| + |\mathcal{V}| + |\mathsf{Enc}_K(\mathcal{D})|)$, where $|\mathsf{Enc}_K(\mathcal{D})|$ is the size of the entire encrypted document corpus, $|\mathcal{T}|$ is the size of the inverted index and $|\mathcal{V}|$ is the size of searchable part. The size of the encrypted inverted index is in $O(N)$, where $N = \sum_{w_i \in W} \mathsf{DB}(w_i)$, refer to Cash et al. [37] construction. Moreover, since m and t can be considered to be a fixed in our scenario ($m \sim 40$), the size of the searchable part is in $O(|\mathrm{W}|)$. Consequently, the total storage complexity, excluding the files, is in $O(N)$, as $N \gg |\mathrm{W}|$.

4.3.3.2 Search phase

SED-1 offers two different types of queries: *wildcard query* and *general query*. We first consider the wildcard query setting. This construction takes into account that wildcard characters (represented by "*" symbol) can be at the beginning of a string or between two strings. Since we are dealing with substring matching, wildcards at the end of the query string are meaningless.

The query is constructed as follows. Let $s \in \mathrm{l} \bigcup \{*\}$ be the string being searched for that includes letters and "*". The string s is associated with the set of letter indexes M_s. For instance, if the user searches for all documents containing the following sequence "$***l^1***l^2**l^3$**", the string s is simplified to $l^1***l^2**l^3$ where $l^i \in \mathrm{l}$ and $p = 3$, where p is the number of wildcard symbols at the front of the substring. The string s can have a size at most equal to m. The string is then written as sequence of letters and "*"s such that: $s = (s^1, s^2, \ldots, s^{|M_s|})$ where:

$$s^j = \begin{cases} l_{M_s[j]} & \text{if } s^j \in \mathrm{l} \\ * & \text{otherwise.} \end{cases}$$

The first step consists of computing the orthornormal value for each letter. We can pre-compute and store all the orthonormal letters in a client side inverted index to avoid computing the orthonormal value each time the user wants to search for a string. The storage or the computation of all possible letters is a small task since the size of all possible letters is mostly small, as in the case of ASCII code for example. At the end of this stage, the string will be $s = (u^1, \ldots, u^{|M_s|})$, where u^j

is equal to:

$$u^j = \begin{cases} \mathsf{Ortho}(\chi, l_{M_s[j]}) & \text{if } s^j = l_{M_s[j]} \\ \sum_{i=1}^{t} \mathsf{Ortho}(\chi, l_i) & \text{otherwise,} \end{cases}$$

where $\mathsf{Ortho}(\chi, l_i)$ is the orthonormal letter corresponding to the letter $l_i \in \mathbf{l}$ as generated by the pre-construction (see Section 4.3.2.1).

The last step is to adjust the orthonormal string s by making the location of first wildcard characters meaningful for the server. Indeed, we will see in the next section that the search phase on the server side is based on an inner product computation between each orthonormal letter in the query and the corresponding searchable part position.

In the wildcard query setting, the main idea is to specify the number, p, of wildcards in the beginning of the query. We can then send to the server the position of the first letter of the string s in the keyword as (s, p), p defining the position at which the server will start the search. In the general query setting the user wants to search for a string pattern regardless of its position. This case is a generalization of the wildcard query where p can have any value in the range $(1, \ldots, m - |s|)$. In this case, the user has only to send the string s to search for and the server will perform a loop to search for all matching keywords.

4.3.3.3 Query phase

The query phase is based on an inner product computation between each element of s and the corresponding searchable part. A keyword w_j matches the substring in the query iff in all positions of the resulting inner products, the values are different from zero. In the following, we detail step by step how the document retrieval is performed in the server side for both kind of queries.

If the server receives the query (s, p), the server assumes that the user has posed a wildcard query. Given that the position of the first relevant letter is $p + 1$, the server first computes an inner product between the first component of s and the corresponding vector v_{p+1} that represents the searchable part stored in the server associated with the position $p + 1$ such that $\alpha_1 = {s_1}^\top \cdot v_{p+1}$ with α_1 being a vector of size equal to $|\mathbf{W}|$. If $\alpha_{1,j} = 0$, where $1 \leq j \leq |\mathbf{W}|$, the keyword w_j does not match the query; otherwise, if $\alpha_{i,j} \neq 0$ it implies that the keyword w_j contains the letter s_1 at position $p + 1$. Indeed, due to the orthogonality of the vectors associated with the letters, when

there is a zero value in a given position in α_1, it means that the letter searched for and the letter existing at this position are different.

A naïve approach to test for this is to compute α_i for $1 \leq i \leq |M_s|$ and verify at the end if there are positions for all these vectors where the values are all different from zero. This method induces many unnecessary inner products since we are computing the inner product even for keywords that we know will not be matching the query. Indeed, if there is a position in α_1 which is equal to zero, we know that from the first verification we can filter out keywords that do not match the query. Instead, we instantiate a set P_0 equal to $(1, \ldots, |\mathbf{W}|)$. While computing the inner product α_1, we delete from P_0 all positions where α_1 is equal to zero and output the new value of positions P_1. Restricted only to the positions P_1, we compute the inner product between s_2 and v_{p+2} so the output of the second search will be a set of positions P_2 which is included in P_1. For α_3 we use P_2 and so on and so forth. We reiterate the same $|M_s|$ operations and the final output will be a set of positions $P_{|M_s|}$.

This resulting set contains the positions of keywords matching the query. The server sends back this set to the client. The client generates a token for every position in the set using the encrypted structure $\mathcal{T}$. For every token, the server queries the structure $\mathcal{T}$ and sends the results to the client. We should underline the fact that the client can find in the final result redundant documents' identifiers. This holds as different keywords can be associated to the same document's identifier in $\mathcal{T}$. To avoid this repetition, a disjunctive SSE construction has to be used instead of the one by Cash et al. [37], we can instantiate a disjunctive SSE using the scheme [82].

Our second case consists of receiving the string s from the user and the server performing a general substring matching. In order to perform this task, the server executes the same operation as if it receives a string with a position information, (s, i), where $1 \leq i \leq m - |s|$. The complexity of our scheme will increase with a multiplicative coefficient equal to $m - |s|$ where m is the size of the longest keyword.

The search computation is not dependent on the size of the database (number of files stored). Rather the search is based on the inner product operations that are in $O(|\mathrm{W}|)$ i.e. linear in the

number of the keywords. Note that since $|\mathrm{W}| \ll n$, we consider the search complexity of SED-1 sub-linear in the size of dataset.

In the next section, we discuss some attacks on our first scheme that will lead us to construct a more secure scheme – SED-2.

4.3.3.4 Problems with SED-1

SED-1 has some vulnerabilities that make it unsuitable for a real-world deployment and which is why we progressively refine SED-1 into SED-2. SED-1 is based on the searchable part $\mathcal{V} = (v_1, \ldots, v_m)$ where the vector v_i contains all letters in the ith position; this searchable part can disclose many information about the distribution of letters in these vectors due to the orthogonal feature of their components. These attacks are analogous to attacks on substitution ciphers using the frequency analysis technique. We discuss two off-line attacks, where the attacks rely only on the knowledge of the searchable part.

Letter frequency attack with known dictionary. We assume that the adversary knows the dictionary W, and the adversary has access to the searchable part $\mathcal{V}$. The goal of the adversary is to find the keywords based on the distribution of letters in the given dictionary. For illustration purposes, we assume that we work with the British Oxford English dictionary. The adversary will launch the attack shown in Algorithm 1 on $\mathcal{V}$.

The algorithm first computes the number of characters stored in the searchable part (lines 3 – 10). The set G_1 contains the occurrence of the first character, G_2 the second, etc. Then line 11 determines the set G_z containing the don't care component; this can be determined by computing the set that contains the maximum number of tuples (i, j) for a given position, where i is the position of the keyword and j is the character in that position. Lines 12 – 14 compute the frequency of resulting characters by calculating first the total number of characters (excluding those in G_z) and then by dividing every set size $|G_i|$ by the total size. Lines 15 – 23 replace the initial values of the searchable part by the corresponding characters based on comparing a known frequency as the one from British Oxford English and the computed ones. Finally, the algorithm outputs $\mathcal{V}$.

Algorithm 1: Letter frequency attack with known dictionary

Data: Searchable part $\mathcal{V}$, dictionary W and characters $c_1, \ldots, c_{p-1}$ of the dictionary language.
Result: disclosed $\mathcal{V}$
Creates p sets of tuples $G_1, \ldots, G_p$ such that
$G_i \leftarrow \{(1,1),(1,2),\ldots,(1,|\mathrm{W}|),\ldots,(m,1),\ldots,(m,|\mathrm{W}|)\}$;
for $i \leftarrow 1$ **to** $p-1$ **do**
  for $j \leftarrow 1$ **to** $initial\ size\ of\ G_i$ **do**
    if $v_{G_i[1]} \neq v_{G_i[j]}$ **then**
      $G_i \leftarrow G_i \setminus \{G_i[j]\}$;
    end
  end
  $G_{i+1} \leftarrow G_{i+1} \setminus G_i$;
end
Determine z such that G_z contains the max of (m,i) where i in the range 1 **to** |W|,;
Compute $totalSize$ the size of all elements within $G_1, \ldots, G_p$ except G_z;
foreach $i \in (1,\ldots,p)$ **do** $F_i \leftarrow \frac{|G_i|}{totalSize}$;
Compute characters' frequency of $c_1, \ldots, c_{p-1}$ in W, output $F'_1, \ldots, F'_{p-1}$;
for $i \leftarrow 1$ **to** p *and* $i \neq z$ **do**
  Find k such that $F'_k = F_i$;
  for $j \leftarrow 1$ **to** $|G_i|$ **do**
    $v_{G_i[j] \leftarrow c_k}$
  end
end
for $j \leftarrow 1$ **to** $|G_z|$ **do**
  $v_{G_i[j] \leftarrow\ blank}$
end
Output $\mathcal{V}$;

The attack is a direct consequence of the detailed searchable part construction that gives away different information about the frequency of keyword letters. Based on $\mathcal{V}$, the attack first determines positions that contain the same letter. Next, the adversary determines the set G_z that contains the ς value that we have used as padding in case of short keywords. This can be done by taking into account the distribution of the keywords' lengths in the dictionary (see figure 4.7) . Thus, based on the frequency of the characters in the dictionary W (see figure 4.6), the adversary can associate the character c_i with its corresponding position.

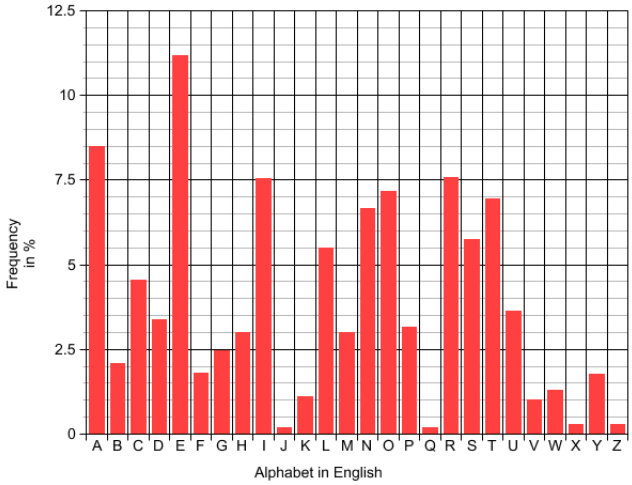

Figure 4.6: Letter Frequency

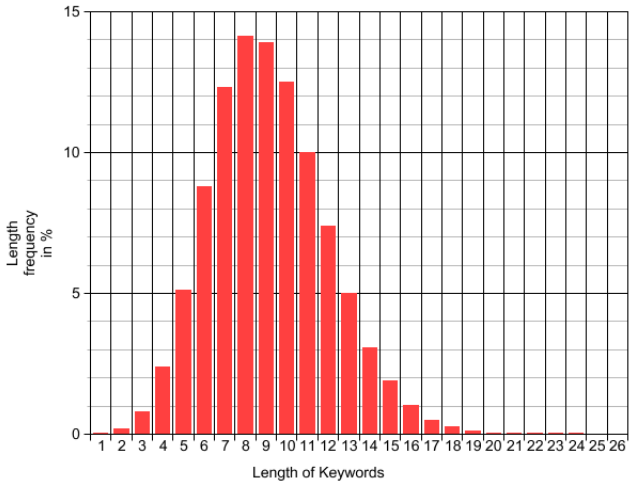

Figure 4.7: Length Frequency

Note that, in this attack, we have made a strong assumption that the adversary has an accurate prior knowledge of the dictionary used by the user. In the next attack, we will assume that the adversary proceeds with the attack without any prior information about the dictionary.

Letter frequency attack with unknown dictionary. The adversary may not know the exact dictionary but can easily know the distribution of characters in the searchable part $\mathcal{V}$ by performing multiple inner products between the different components of these vectors. The adversary can then have a finite size of possible characters, which we denote by q. By looking for all possible combinations of letters, the adversary performs $q!$ combinations to determine if for each combination all keywords belong to some language. The brute force can be expensive. For example, if the number of characters $q = 26$ (which is the size of the English alphabet), the possible solutions are approximately equal to 2^{88} (26!). However, the adversary can make the search cheaper without the dictionary.

The attacker can try to use certain types of information to refine the data distribution model. These can be, for example, the geographic localization of the user, the field of interest, etc. However, we assume these to be well hidden from the adversary. Moreover, knowing the exact distribution of characters for each keyword in $\mathcal{V}$, the adversary can generate therefore the frequency of letters in the whole dataset. In the state of the art, there are many frequency studies about the distribution of letters that differs depending on the field such as Scientific, Religious, Financial field etc. Moreover, the adversary can go further by computing the frequency of letters at the beginning of the word, the end of the word, length distribution, frequency of bigrams, trigrams etc., that give more accurate idea of the language used and consequently approach the adversary from the exact letters.

4.3.4 Substring Search Over Encrypted Data #2 – SED-2

We have seen that the first scheme SED-1 is insecure and can leak the keywords in the case of a known dictionary. We should point out that it is a real challenge to perform substring matching and at the same time hide the distribution of letters within the dictionary from the server. In this section,

we present the second and main construction, SED-2, that *partially*[5] hides the letter-distribution of keywords.

The main challenge for SED-2 is to maintain the order of letters within a given keyword. Indeed, only summing up normalized letters does not save any pattern with a size bigger than that of the keyword. SED-2 addresses this issue by creating a sort of chain of letters, intelligible to the user with an a-prior stored state, that allows one to search for any pattern. Add to that, this new feature SED-2 also consists of linking every character to its position. That is, two equal characters at different positions will be treated in the setup phase as two totally different characters. This feature enables to hide a great part of the information leakage in SED-1. We will also see that this position based construction can be further generalized with a pattern construction to even hide more about characters' frequency, refer to Section 4.3.6. SED-2 is more efficient and induces less storage complexity than SED-1. We present the detail of SED-2 in Algorithm 4.8.

4.3.4.1 Correctness of SED-2

The correctness of the scheme is shown quite easily. The query will contain orthonormal letters depending on the position in the string searched for. These orthonormal letters are multiplied by random elements and divided by the sum of elements of the chain a_i depending on the position as well. While the server receives the query, the inner product of the query and the component $\mathcal{V}_i$ will be equal to one if and only if all letters of the query exist in the same position in the coded component $\mathcal{V}_i$. If this is the case the inner product of each equal orthonormal letters will be $\sum_{i=1}^{|M_s|} a_{M_s[i]}$ where s is the substring searched for, which is equal to the denominator of the query. If this product is equal to one, the server will send the exact corresponding documents that match the substring from the inverted index.

4.3.5 Security analysis for SED-2

4.3.5.1 Leakage description of SED-2

SED-2 construction, as any structured encryption construction, has some leakage. This leakage is divided in two types. A *setup leakage* that the server (adversary) can undercover by just looking

[5]Refer to Section 4.3.5.1 for exact description of SED-2 leakage.

Let consider that $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Token})$ be an inverted index structured encryption scheme. Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ a semantically secure symmetric encryption scheme. We define $\mathsf{SED} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$, the substring search over encrypted data as follows:
$\underline{\mathsf{Setup}(1^k, \mathcal{D})}$:

1. consider the set of characters $\mathbf{l} = \{l_1, \cdots, l_t\}$ and compute $L = (\mathbf{l^1}, \cdots, \mathbf{l}^m)$, where $\mathbf{l}^i = \{l_1^i, \cdots, l_t^i\}$ and $l_j^i = l_j \| i$, for all $i \in [m]$ and $j \in [t]$;
2. compute $(K, \mathcal{T}) \leftarrow \mathsf{STE.Setup}(1^k, \mathsf{DB})$, where DB is the inverted index indexing $\mathcal{D}$;
3. set key $K = (K_1, K_2, \chi)$ such that $\chi \leftarrow \{0,1\}^k$ and $K_2 \leftarrow \mathsf{Gen}(1^k)$;
4. compute $(u_1^1, \cdots, u_t^1, u_1^2, \cdots, u_t^2, \cdots, u_1^m, \cdots, u_t^m) \leftarrow \mathsf{Ortho}(\chi, L)$;
5. compute $\kappa = (a_1, \ldots, a_m) \xleftarrow{\$} \{0,1\}^{k \cdot m}$, where $\forall i, j \; a_i \neq a_j$. Compute then $\mathcal{V}_j = \sum_{i=1}^{|w_j|} a_i \cdot u^i_{M_j[i]}$, for all $j \in [|\mathrm{W}|]$ such that:
$$\mathcal{V} = (\sum_{i=1}^{|w_1|} a_i \cdot u^i_{M_1[i]}, \ldots, \sum_{i=1}^{|w_{|\mathrm{W}|}|} a_i \cdot u^i_{M_{|\mathrm{W}|}[i]});$$
6. compute the encryption of all documents $\mathsf{Enc}_{K_2}(\mathcal{D}) = (\mathsf{Enc}_{K_2}(\mathcal{D}_1), \cdots, \mathsf{Enc}_{K_2}(\mathcal{D}_n))$;
7. output $(K, \kappa, \mathsf{EDB})$, where $\mathsf{EDB} = (\mathsf{Enc}_{K_2}(\mathcal{D}), \mathcal{T}, \mathcal{V})$.

$\underline{\mathsf{Token}(K, \kappa, s)}$:

1. compute M_s, the index positions of substring s and parse $\kappa = (a_1, \cdots, a_m)$;
2. compute $(u_1^1, \cdots, u_t^1, u_1^2, \cdots, u_t^2, \cdots, u_1^m, \cdots, u_t^m) \leftarrow \mathsf{Ortho}(\chi, L)$;
3. output $\mathsf{tk} = \frac{1}{\sum_{i=1}^{|M_s|} a_{M_s[i]}} \sum_{i=1}^{|M_s|} u^i_{M_s[i]}$.

$\underline{\mathsf{Query}(\mathsf{tk}, \mathsf{EDB})}$:

1. initialize an empty set Result and parse $\mathsf{EDB} = (\mathsf{Enc}_{K_2}(\mathcal{D}), \mathcal{T}, \mathcal{V})$;
2. for all $j \in [|\mathrm{W}|]$, if $\mathsf{tk} \cdot \mathcal{V}_j = 1$, then add j to Result. Send Result to the client;
3. for all $j \in \mathsf{Result}$, compute $\mathsf{tk}_j \leftarrow \mathsf{STE.Query}(K_1, j)$. Send $(\mathsf{tk}_j)_{j \in \mathsf{Result}}$ to the server
4. compute for each token $\mathsf{tk} \in (\mathsf{tk}_j)_{j \in \mathsf{Result}}$, $r \leftarrow \mathsf{STE.Query}(\mathcal{T}, \mathsf{tk})$. Send all results r to the client.

Figure 4.8: Substring search over encrypted data $\mathsf{SED} - 2$ for the wildcard search case

to the encrypted data structures, and, a *query* leakage that the adversary can learn during the search phase. SED-2, if compared to single keyword SSE constructions, leaks more at the price of offering better expressiveness. In the following, we details both of these leakages.

Setup Leakage $\mathcal{L}_{\mathsf{S}}$. SED-2 setup phase outputs an encrypted inverted index $\mathcal{T}$, along with the searchable part $\mathcal{V}$. The first encrypted data structure leaks $\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}(\ D)$. Based on the specific instantiation chosen for our algorithm, this leakage might be slightly different from a construction to another. If we choose the construction by Cash et al. [37], then the encrypted inverted index only leaks N, the number of keyword-identifiers pairs such that $N = \sum_{w \in \mathrm{W}} \mathsf{DB}(w)$. Contrary, to previous SSE, $\mathcal{V}$ can leak more than W, it leaks the number of characters and therefore the size of the alphabet used for the construction t. Moreover, the server can learn whether two keywords w_i and w_j have the same character at the same position by performing simple inner product between the coordinates of $\mathcal{V}$. We formalize this leakage in the following:

$$\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}(\mathcal{D}) = \{\mathcal{L}_{\mathsf{S}}^{1}(\mathcal{D}), \mathcal{L}_{\mathsf{S}}^{2}(\mathcal{D})\},$$

where $\mathcal{L}_{\mathsf{S}}^{1}(\mathcal{D}, \mathrm{W}) = (\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}(\mathcal{D}), m, |\mathrm{W}|, t)$ and $\mathcal{L}_{\mathsf{S}}^{2}(\mathcal{D}) = \{i \mid \forall i \in [m], v_i = w_i\}_{v,w \in \mathrm{W}}$.

Query Leakage $\mathcal{L}_{\mathsf{Q}}$. During search operations, SED-2 like most of SSE constructions, leak the search pattern and access pattern, given τ queries $(q_i)_{i \in [\tau]}$. However, as previously illustrated, due to the additional expressiveness, SED-2 can leak much more than traditional exact keyword SSEs, $\mathcal{L}_{\mathsf{Q}}^{\mathsf{ste}}(\mathcal{D}, (q_i)_{i \in [\tau]})$. In SED-2, for the wildcard search, the server learns whether two queries have the same character at the same position. This leakage, dubbed as *query composition* leakage, can give the adversary an additional knowledge about the searched for queries. It is clear that the query composition leakage implicitly encapsulates the query occurrences which is the traditional search pattern leakage of SSE constructions. The second component of the adaptive leakage is the one capturing the association between the query and the keywords in $\mathcal{V}$. Every searched for keyword will be associated to some specific keywords in $\mathcal{V}$, we refer to this leakage as *keywords pattern* leakage. Finally, the server has to retrieve the identifiers of documents that match a specific query. That is, the server will learn the association between every query and the documents' identifiers,

which is known in SSE's literature as the *access pattern*. We formalize the adaptive leakage in the following:

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{sed}}(\mathcal{D},(q_i)_{i\in[\tau]}) = \{\mathcal{L}_{\mathsf{Q}}^{\mathsf{ste}}(\mathcal{D},(q_i)_{i\in[\tau]}), \mathsf{QC}(\mathcal{D},(q_i)_{i\in[\tau]}), \mathsf{KP}(\mathcal{D},(q_i)_{i\in[\tau]}), \mathsf{AP}(\mathcal{D},(q_i)_{i\in[\tau]})\},$$

where

$$\mathsf{QC}(\mathcal{D},(q_i)_{i\in[\tau]}) = \{A \mid \forall i,j \in [\tau],\ \forall l \in [m],\ A[i,j,l] = 1 \text{ if } q_{i,l} = q_{j,l}\},$$

and,

$$\mathsf{KP}(\mathcal{D},(q_i)_{i\in[\tau]}) = \{A \mid \forall i \in [\tau],\ A[i] = (i_1,\cdots,i_h),\ \text{where } \forall j \in [h], \forall l \in [|q_i|],\ q_{i,l} = w_{i_j,l}\},$$

and,

$$\mathsf{AP}(\mathcal{D},(q_i)_{i\in[\tau]}) = \{A \mid \forall i \in [\tau], A[i] = (D(w_{i_1}),\cdots,D(w_{i_h})),\ \text{where } (i_1,\cdots,i_h) = \mathsf{KP}(\mathcal{D},q_i)\}$$

This setup and query leakage described above is the only leakage that SED-2 leaks to the adversary when a search is performed. We will show in the next section that SED-2 is $(\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{sed}})$-semantically secure as in Definition 2.3.3, i.e., that SED-2 does not leak any information to the server besides the stateful captured leakage $\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}$ and $\mathcal{L}_{\mathsf{Q}}^{\mathsf{sed}}$. Note that the leakage in some scenarios might help the adversary to infer additional information about the encrypted data structure, especially when the adversary holds adequate auxiliary information.

4.3.5.2 Security proof of SED-2

The SED-2 protocol is $(\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{sed}})$-semantically secure in the honest but curious model. We now provide a simulation based proof in support of our security claim. We restrict our proof to the wildcard search model.

Theorem 4.3.1. *If* SKE *is a semantically-secure encryption scheme, G is a pseudo-random generator,* STE *a $(\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{ste}})$-semantically secure inverted index encryption, then SED-2 is $(\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{sed}})$-semantically secure.*

Proof. Let $\mathcal{S}^{\mathsf{ste}}$ be the simulator guaranteed to exist by the semantic security of STE. Let us consider the simulator $\mathcal{S}$ of SED that works as follows.

Given $\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}(\mathcal{D})$, $\mathcal{S}$ simulates the output of the Setup as follows:

1. set $\mathcal{T} \leftarrow \mathcal{S}^{\mathsf{ste}}(\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}(\mathcal{D}))$

2. based on $\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}$, the simulator $\mathcal{S}$ extracts $|\mathrm{W}|$, t and m. $\mathcal{S}$ generates $\chi \xleftarrow{\$} \{0,1\}^k$. The simulator computes $(g_1, \cdots, g_{t \cdot m}) \leftarrow G(\chi)$, where $|g_i| = t \cdot m + k$, where k is the security parameter. For every $i \in [|\mathrm{W}|]$, the simulator, based on $\mathcal{L}_{\mathsf{S}}^2(\mathcal{D})$, computes $\mathcal{V}_i = (g_j)_{j \in A_i}$, where $A_i = \mathcal{L}_{\mathsf{S}}^2(\mathcal{D})[i]$.

3. the simulator $\mathcal{S}$ generates a key at random $K_2 \xleftarrow{\$} \{0,1\}^k$ and sets

$$\mathsf{Enc}_{K_2}(\mathcal{D}) = (\mathsf{Enc}_{K_2}(\mathbf{0}^{|D_1|}), \cdots, \mathsf{Enc}_{K_2}(\mathbf{0}^{|D_n|}))$$

4. The simulator outputs $\mathsf{EDB} = (\mathsf{Enc}_{K_2}(\mathcal{D}), \mathcal{V}, \mathcal{T})$.

The simulator now simulates the token tk for the wildcard search as follows. $\mathcal{S}$ has as input $\mathcal{L}_{\mathsf{Q}}^{\mathsf{sed}}$. Given the queries $(s_i)_{i \in [\tau]}$, the simulator generates for every $i \in [\tau]$, tk_i such that for all $j \in [|s_i|]$, if $\exists z < i$ such that $\mathsf{QC}[z, i, j] = 1$, then $\mathsf{tk}_{i,j} = g$, where g is the previously generated value, otherwise if $\mathsf{QC}[z, i, j] = 0$, the simulator sets $\mathsf{tk}_{i,j} = g$, where g is sub-vector of $G(\chi)$ that verifies $g \in \mathcal{V}_{j_1} \cap \cdots \cap \mathcal{V}_{j_h}$, where $(j_1, \cdots, j_h) \in \mathcal{L}_{\mathsf{S}}^2(\mathcal{D})$

The simulator $\mathcal{S}$ generates the tokens for $\mathcal{T}$ such that for each $i \in [\tau]$, for all $j \in \mathsf{KP}[i]$, set $\mathsf{tk}_{i,j} \leftarrow \mathcal{S}^{\mathsf{ste}}(\mathcal{L}_{\mathsf{Q}}^{\mathsf{ste}}(\mathcal{D}, i)))$.

It remains to show that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability that $\mathbf{Real}_{\mathsf{SED},\mathcal{A}}(k)$ and the probability that $\mathbf{Ideal}_{\mathsf{SED},\mathcal{A},\mathcal{S}}(k)$ output 1 is negligibly close. We show this by the sequence of these hybrid games:

- Game_0 is the same as $\mathbf{Real}_{\mathsf{SED},\mathcal{A}}(k)$ experiment. Create an empty dictionary T.

- Game_1 is the same as Game_0 except that every component of $\mathcal{V}$ is replaced with the original (non-orthogonal vectors) PRG evaluations. Every component $\mathcal{V}_i$ in $\mathcal{V}$ is replaced with a vector of the corresponding PRG evaluations of characters composing $\mathcal{V}_i$, for all $i \in [|\mathrm{W}|]$. Also, for all $i \in [\tau]$, tk_i is replaced with the vector of PRG evaluation of the characters searched for.

- Game_2 is the same as Game_1 except that the values of PRG evaluation are replaced with random values as follows: for all $i \in [|\mathsf{W}|]$, for all $v \in \mathcal{V}_i$, if $\mathsf{T}[v]$ is not empty, then set v to $\mathsf{T}[v]$, otherwise, set $\mathsf{T}[v] \xleftarrow{\$} \{0,1\}^k$, then set v to $\mathsf{T}[v]$, and update $\mathcal{V}_i$ accordingly. The queries tk_i, for all $i \in [\tau]$, are replaced with random values as follows: for all $i \in [\tau]$, for all $v \in \mathsf{tk}_i$, $\mathsf{T}[v]$ is not empty, then set v to $\mathsf{T}[v]$, otherwise, set $\mathsf{T}[v] \xleftarrow{\$} \{0,1\}^k$, then set v to $\mathsf{T}[v]$, and update tk_i accordingly.

- Game_3 is the same as Game_2 except that the encryption of documents is replaced with $\mathsf{Enc}_{K_2}(\mathbf{0}^{|D|})$, for all $D \in \mathcal{D}$.

- Game_4 is the same as Game_3 except that $\mathcal{T}$ is replaced with $S^{\mathsf{ste}}(\mathcal{L}^{\mathsf{ste}}_{\mathsf{S}}(\mathcal{D}))$ and for all $i \in \tau$, for all $j \in \mathsf{KP}[i]$, replace all tokens by $\mathsf{tk}_{i,j} \leftarrow \mathcal{S}^{\mathsf{ste}}(\mathcal{L}_{\mathsf{Q}}(\mathcal{D}, i)))$.

Note that Game_4 equals the ideal experiment.

Claim. For all PPT $\mathcal{A}$,

$$\Pr[\mathsf{Game}_1 = 1] = \Pr[\mathsf{Game}_0 = 1],$$

This transition just replaces the orthogonal values in every $\mathcal{V}_i$ in $\mathcal{V}$ and tk_j, for $j \in [\tau]$, by the non-orthogonal values, and represents them as a vector of a PRG evaluation. Recall that this trivially stems from the fact that we assume that the orthogonalization is just a tool to speed-up the verification and there exists a polynomial time algorithm that can perform this basis' transformation[6].

That is the view of an PPT adversary $\mathcal{A}$ in Game_0 and Game_1 is the same.

Claim. For all PPT $\mathcal{A}$,

$$\Pr[\mathsf{Game}_2 = 1] - \Pr[\mathsf{Game}_1 = 1] \leq \mathsf{negl}(k),$$

We show that if there exists an adversary $\mathcal{A}$ that breaks the claim, then there exists an adversary $\mathcal{B}$ that can distinguish between the output of pseudo-random generator and a random value with

[6]This game shows that our SED security is not based on the linear transformation but on the prior PRG evaluation.

respect to a simulator $\mathcal{S}'$. $\mathcal{B}$ emulates $\mathcal{A}$. When receiving, $\mathcal{D}$ from $\mathcal{A}$, $\mathcal{B}$, generates all characters in $\mathcal{D}$, $(l_1, \cdots, l_t)$, computes as in Step 1 in the Setup phase in Algorithm 4.8, L. $\mathcal{B}$ generates an inverted index that associates every keyword w to its documents identifiers. $\mathcal{B}$ outputs $\mathcal{L}$. Upon receiving L^*, from either $\mathbf{Read}_{PRG,\mathcal{B}}(k)$, or $\mathbf{Ideal}_{PRG,\mathcal{B},\mathcal{S}'}$, $\mathcal{B}$ outputs to $\mathcal{A}$: $(\mathsf{Enc}_{K_2}(\mathcal{D}), \mathcal{V}, \mathcal{T})$, such that:

$K_2 \xleftarrow{\$} \{0,1\}^k$, $(K_1, \mathcal{T}) \leftarrow \mathsf{STE.Setup}(1^k, \mathcal{D})$ and for all $i \in |\mathrm{W}|$, for all $v \in \mathcal{V}_i$, if $\mathsf{T}[v]$ is not empty, then set v to $\mathsf{T}[v]$, otherwise, set $\mathsf{T}[v]$ to the next k bits in L^*, then set v to $\mathsf{T}[v]$, and update $\mathcal{V}_i$ accordingly.

If $\mathcal{A}$ outputs a substring s_i for all $i \in [\tau]$, $\mathcal{B}$ outputs tk_i^* such that: for all $v \in \mathsf{tk}_i$, if $\mathsf{T}[v]$ is not empty, then set v to $\mathsf{T}[v]$, otherwise, set $\mathsf{T}[v]$ to the next k bits in L^*, then set v to $\mathsf{T}[v]$, and update tk_i accordingly.

$\mathcal{B}$ sends the following to $\mathcal{A}$: $(\mathsf{tk}_i, \mathsf{tk}_{i,1}, \cdots, \mathsf{tk}_{i,|\mathsf{Result}|})$, where Result is a set containing all indices j of $\mathcal{V}$ where characters of $\mathsf{tk}_i \subseteq \mathcal{V}_j$.

After answering all $\mathcal{A}$'s queries, $\mathcal{A}$ outputs a bit that $\mathcal{B}$ returns as its own output. We want to emphasize that if $\mathcal{B}$ has been executed in a $\mathbf{Real}_{PRG,\mathcal{B}}(k)$, than the view of $\mathcal{A}$ is exactly the one in Game_1, otherwise then the view of $\mathcal{A}$ is the one of Game_2. It follows by our assumption of $\mathcal{A}$ contradicts the claim that:

$$\Pr[\mathbf{Real}_{PRG,\mathcal{B}}] - \Pr[\mathbf{Ideal}_{PRG,\mathcal{B},\mathcal{S}'}]$$

is non-negligible in k which is a contradiction.

Claim. For all PPT adversary $\mathcal{A}$,

$$\Pr[\mathsf{Game}_3 = 1] - \Pr[\mathsf{Game}_2 = 1] \leq \mathsf{negl}(k),$$

This claim clearly holds under the semantic security assumption.

Claim. For all PPT adversary $\mathcal{A}$,

$$\Pr[\mathsf{Game}_4 = 1] - \Pr[\mathsf{Game}_3 = 1] \leq \mathsf{negl}(k),$$

We show that if there exists an adversary $\mathcal{A}$ that breaks the claim, then there exists an adversary $\mathcal{B}$ that can can break the semantic security of the inverted index encryption STE with respect to an arbitrary PPT simulator $\mathcal{S}'$. $\mathcal{B}$ start running $\mathcal{A}$. When receiving, $\mathcal{D}$ from $\mathcal{A}$, $\mathcal{B}$, generates all characters in $\mathcal{D}$, $(l_1, \cdots, l_t)$, computes $\mathcal{L}$. $\mathcal{B}$ generates an inverted index $\mathsf{DB} = (w_i, D(w_i))_{i \in |\mathrm{W}|}$ that associates every keyword w to its documents identifiers $D(w_i)$. $\mathcal{B}$ outputs DB. Upon receiving $\mathcal{T}^*$, from either $\mathbf{Read}_{\mathsf{STE},\mathcal{B}}(k)$, or $\mathbf{Ideal}_{\mathsf{STE},\mathcal{B},\mathcal{S}'}$, $\mathcal{B}$ outputs to $\mathcal{A}$: $(\mathsf{Enc}_{K_2}(\mathcal{D}), \mathcal{V}, \mathcal{T}^*)$, such that:

$K_2 \xleftarrow{\$} \{0,1\}^k$, for all $i \in |\mathrm{W}|$, for all $v \in \mathcal{V}_i$, set v to $\mathsf{T}[v]$. If $\mathcal{A}$ outputs a substring s_i for all $i \in [\tau]$, $\mathcal{B}$ outputs tk_i such that for all $v \in \mathsf{tk}_i$, v to $\mathsf{T}[v]$ and outputs Result as its own answer to $\mathcal{T}^*$ and receives $(\mathsf{tk}^*_{i,1}, \cdots, \mathsf{tk}^*_{i,|\mathsf{Result}|})$, where Result are all indices j of $\mathcal{V}$ where characters of $\mathsf{tk}_i \subseteq \mathcal{V}_j$.

$\mathcal{B}$ sends the following to $\mathcal{A}$: $(\mathsf{tk}_i, \mathsf{tk}^*_{i,1}, \cdots, \mathsf{tk}^*_{i,|\mathsf{Result}|})$.

After answering all $\mathcal{A}$'s queries, $\mathcal{A}$ outputs a bit that $\mathcal{B}$ returns as its own output. We want to emphasize that if $\mathcal{B}$ has been executed in a $\mathbf{Real}_{\mathsf{STE},\mathcal{B}}(k)$, than the view of $\mathcal{A}$ is exactly the one in Game_3, otherwise if $\mathcal{B}$ has been executed in $\mathbf{Ideal}_{\mathsf{STE},\mathcal{B},\mathcal{S}'}(k)$, then the view of $\mathcal{A}$ is the one of Game_4. It follows by our assumption that:

$$\Pr[\mathbf{Real}_{\mathsf{STE},\mathcal{B}}] - \Pr[\mathbf{Ideal}_{\mathsf{STE},\mathcal{B},\mathcal{S}'}]$$

is non-negligible in k which is a contradiction. This ends our proof.

■

4.3.6 SED-2 Generalization

SED-2 hides information related to characters that exist at different positions while not impacting the scheme's correctness. However, we have described in Section 4.3.5.1 SED-2 leakage, and we have shown that an adversary can recover non-trivial information about the encrypted data structure even in the setup phase. With adequate auxiliary information, an adversary can run many attacks based on characters frequency. This unfortunately can lead to the disclosure of all part of keywords dictionary. However, we can generalize SED-2 construction that leaks much lesser with a new pattern's construction. The ultimate goal of the generalized SED-2 construction is to decrease the setup leakage to be as insignificant as possible.

4.3.6.1 Overview

The setup phase is based on a new parameter, pattern length γ, that determines the length of the pattern that is going to be orthogonalized. Every pattern is now determined by its position but also by its length. For every character in $\mathbf{l} = \{l_1, \cdots, l_t\}$, for all $i \in [m]$, add

$$\underbrace{c_1 \| (i-\gamma), \cdots, c_\gamma \| (i-1)}_{\gamma \text{ characters}}, l \| i, \underbrace{c_{\gamma+1} \| (i+1), \cdots, c_{2\gamma} \| (i+\gamma)}_{\gamma \text{ characters}}$$

to L, for all $c_1, \cdots, c_{2\gamma} \in \mathbf{l}$ and $\gamma \in [m]$. Note that SED-2 is a special case of our generalization when $\gamma = 0$.

The next step is to apply the orthogonalization process Ortho that outputs orthonormal vectors.

The way we construct the searchable part $\mathcal{V}$ is similar to the one in SED-2. For all $w \in \mathrm{W}$, we first determine all patterns in w, and sums up all the corresponding orthonormal characters while using the chaining between patterns as in SED-2. This pattern enables to search for all substrings that have a γ-length pattern at least. The search is very similar to SED-2.

This generalization introduces a new balance between search expressiveness and leakage. In SED-2, the leakage can be greatly reduced when increasing γ. We notice that starting from $\gamma = 1$, the setup leakage dramatically reduces. However, note that this comes also at the expense of a slightly larger components of the searchable part $\mathcal{V}$.

The setup leakage as well as the adaptive leakage is now limited to the following. The encrypted searchable part now leaks only the information that two keywords have the same pattern if they are also at the same position. For example, the keywords *arbitrary* and *binary* both contain the pattern ary, but at two different positions. This will translate in two different orthonormal values that are not going to be disclosed in the setup phase. Another example reduced leakage is the following. Consider the keywords *art* and *arc*, they both contain two characters at the same positions. SED-2 would have disclosed this information in the setup phase while in the generalization this information is hidden. Recall that SED-2 is a special case where the length of the pattern equals 0. In the following, we formalize the leakage of the generalized construction of SED-2.

4.3.6.2 Leakage

As illustrated in the previous section, the generalization offers better security guarantees at the cost of slightly reduced expressiveness. We provide in the following a formal description of setup leakage. The adaptive leakage is similar to the one described for SED-2.

Setup Leakage $\mathcal{L}_{\mathsf{S}}^{\mathsf{sed}}$.

$$\mathcal{L}_{\mathsf{S}}(\mathcal{D}) = \{\mathcal{L}_{\mathsf{S}}^{1}(\mathcal{D}), \mathcal{L}_{\mathsf{S}}^{2}(\mathcal{D})\},$$

where $\mathcal{L}_{\mathsf{S}}^{1}(\mathcal{D}) = (\mathcal{L}_{\mathsf{S}}^{\mathsf{ste}}(\mathcal{D}), m, |\mathrm{W}|, q, (|D_i|)_{i\in[n]})$,

and

$$\mathcal{L}_{\mathsf{S}}^{2}(\mathcal{D}) = \{j \mid \forall j \in [m], \forall i \in [\gamma], v_{j+i} = w_{j+i} \text{ and } v_{j-i} = w_{j-i}\}_{v,w\in\mathrm{W}}.$$

Note that $\mathcal{L}_{\mathsf{S}}^{2}(\mathcal{D})$ is a subset of the one of the first version of SED-2. This underlines the fact that the generalized version of SED-2 leaks much lesser when compared to SED-2. Note that the security proof of our proposed construction is very similar to the one detailed for SED-2.

4.3.7 Performance Analysis

We have evaluated the performance of SED-2. For this purpose, we developed a proof-of-concept using Java. We use the Enron document corpus that has over 500,000 emails and is about 1.5 Gbytes in size. Our experiments clearly demonstrate the efficiency of the SED-2 protocol. We discuss the experiments and results in more details below.

4.3.7.1 Experiment setup

We executed a number of experiments to validate not only the correctness of the theoretical complexities, but also study the deployability of SED-2 in a realistic setting. Our analyses include a quantification in terms of execution time and storage for the complete SED-2 protocol. In addition to the construction of the inverted index and the searchable part, we study the overhead related to the initial stages of the protocol, namely, determining unique keywords for every document as well as creating a dictionary that is associated with this set of documents. Any encrypted search scheme will involve these steps and execution time for this phase is far from negligible.

We chose the Enron corpus [34] for our performance studies. The corpus contains more than 500,000 emails from about 150 users of the erstwhile Enron Corporation. The total size of the document corpus is nearly equal to 1.5 Gbytes. The high number of unique keywords in the corpus was the main reason behind our choice of the Enron corpus since the search complexity of SED-2 is dependent on the size of the dictionary and not on the number of the outsourced files. We do not consider the overhead due to the encryption of files in our performance analysis.

The SED-2 scheme is implemented in Java using the JSci library [11] for keywords orthogonalization. Since the components of the searchable part are decimal numbers, we use Java's BigDecimal class for accurate precision. (For example, if the output is equal to one plus or minus 10^{-10} we assume that there is a match.) We also use Java's SecureRandom class to generate random numbers in the query and searchable part construction. The experiments have been executed on a 8-core Intel Core i7-3630QM CPU 2.40GHz with 8 Gbytes of RAM. For execution time, the values displayed in the graph are the sample mean values over several independent executions.

4.3.7.2 Results

We first show in figure 4.9 the execution time needed to determine for every file, the number of unique keywords and then constructing the entire dictionary of all these files (graph labeled as "keywords indexation"). Based on unique keywords associated with each file, we construct the inverted index that links each keyword to all documents that contain the keyword. These phases can be merged to gain time, however they are still linear in the number of files (multiplied by the number of keywords in the case of textual files, which was our case in the Enron emails).

SED-2 complexity is dependent on the size of the dictionary, we show in figure 4.10 that the number of unique keywords associated with a number of Enron emails. We notice that for 90,000 emails, the number of unique keywords is greater than the size of the Enable (172,820 keywords) or the British Oxford (127,238 keywords) dictionaries. This is due to the additional scientific or organizational notations, proper names etc., in the ENRON corpus and also due to the fact that the other dictionaries contains only the base words and not their variants like the plurals or so.

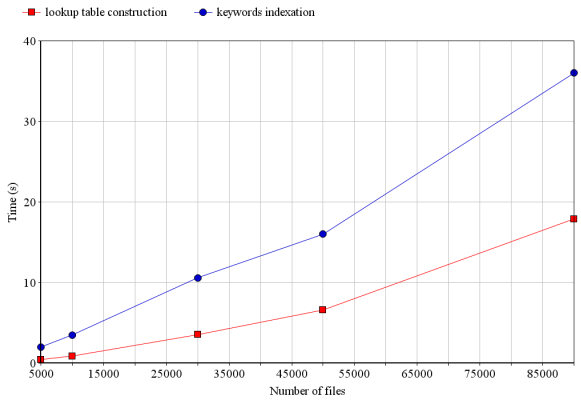

Figure 4.9: Time of inverted index encryption

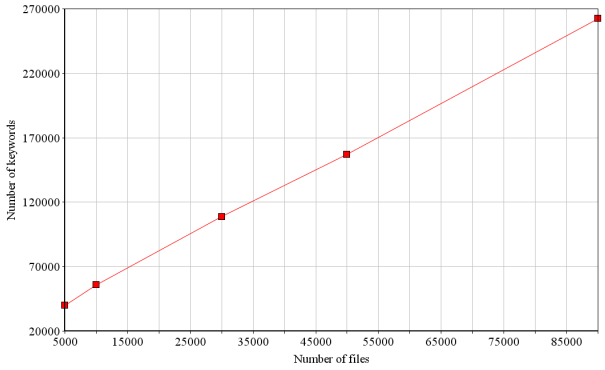

Figure 4.10: Number of unique keywords

The setup phase of SED-2 pre-construction is based on the creation of a set of letters with a size equal to the initial set of letters, times the number of possible positions. The number of possible positions is equal to the size of longest keyword. We have assumed a maximum size equal to 40, and the set of initial letters equal to the ASCII representation. The final size of the normalized letters is equal to 9.2 MBytes constructed in 2.4 seconds. Once the normalized letters are created, they are stored in the client side for more efficiency during the search phase.

Figure 4.11 shows the time to construct the searchable part as a function of the number of unique keywords based on the dictionary. The study clearly demonstrates the linearity of the searchable part of the construction. As indicated earlier, the searchable part construction is independent of the number of files in the corpus. In figure 4.12 we show the size of the inverted index and the searchable part that the server should store dependent on the number of unique keywords indexed.

During the search phase, the user a-priori loads the normalized letters in memory to avoid I/O overhead. The time to generate the search query construction is equal to 78 micro seconds. The test phase takes as an input the searchable part, the inverted index and outputs the corresponding set of documents, Figure 4.13 shows the time of search in one thread by process and parallelized multiple threads in one process.

The server does not have to wait until the end of the search phase to send the matching encrypted documents, since the searchable part is linear. Once the server finds a matching it sends

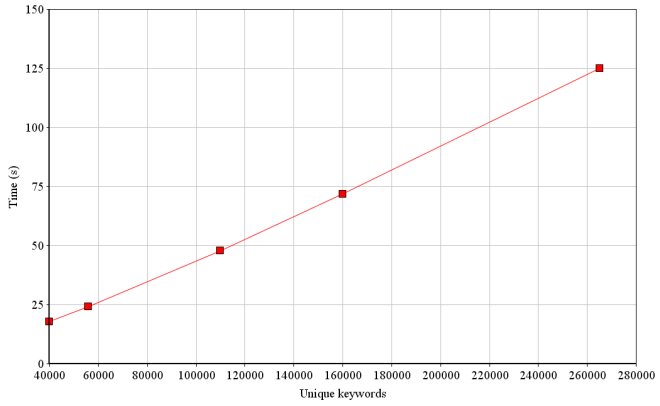

Figure 4.11: Searchable part construction

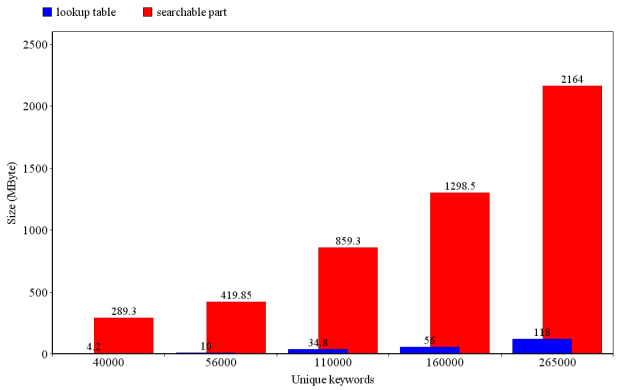

Figure 4.12: Size of the encrypted data structure

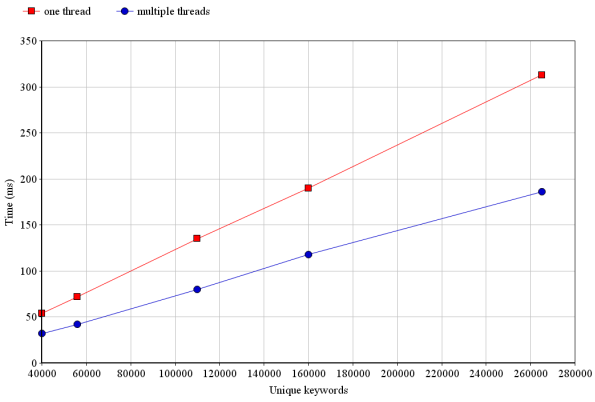

Figure 4.13: Search phase time

directly the encrypted document to the user. Based on the graph, the mean search time per keyword is equal to 0.85 micro seconds for parallel search and 1.3 micro seconds for one thread process.

To enhance the search phase and reduce communication overhead, the server can first send the matching documents' identifiers to the user so that the user can chose the documents that he is interested in. The server can then send these documents. We can enhance this by giving the user an overview of the documents with the document identifiers. However, this will require the user to encrypt a small part of each file during the Setup phase, that it will later be sent to the user with the document identifiers.

Section Conclusion. We presented in this section SED a substring STE construction that has an overhead sub-linear in the size of the data set when considering that $|\mathrm{W}| \ll n$. We have proposed two incremental constructions with a very well described leakage. We believe that the generalized version of SED-2 is the most suitable for substring search scenarios as it has a better balance between security, expressiveness and efficiency. We want, however, to point out that the recent work by Chase and Shen [44] has better leakage with logarithmic search efficiency. The work by Faber et al. [59] is similar to our SED-2 generalization but with a better leakage profile. Moataz and Blass proposed recently an oblivious substring search that provides the best security guarantees for substring search but at the cost of poly-logarithmic blowup for the search overhead. Nevertheless, we want to point out that it would be very interesting to investigate the leakage of substring STE constructions for a better understanding. At this point, we are unaware of the right balance between leakage disclosure, efficiency and expressiveness. We believe that an interesting open problem in SSE would be to investigate this balance and its impact on SSE constructions.[7]

[7] Seny Kamara is working on a theoretical framework of leakage in general, I am also aware that he is studying the importance of decreasing the leakage in SSE construction.

Chapter 5

Oblivious RAM

> The only way of finding the limits of the possible is by going beyond them into the impossible.
>
> *Arthur C. Clarke*

Chapter Outline. In this chapter, we provide all technical details of our ORAM contributions:

1. resizable ORAM in Section 5.1. We are the first to introduce the concept of *resizability* in ORAM, especially, in the case of tree-based ORAM. We particularly show that increasing arbitrary the size of the tree might be harmful for security reasons. We therefore introduce provable techniques that show how to securely increase/decrease the size of an ORAM.

2. r-ORAM, a recursive ORAM in Section 5.2, we show that traditional binary (or κ-ary) tree structure in tree-based ORAM does not represent the optimal structure to use in an ORAM. We introduce new recursive structure that further reduces the ORAM asymptotics.

3. C-ORAM, a constant communication ORAM in Section 5.3. This work represents, one of the most optimal ORAM in a single-server setting in terms of communication overhead. This work leverages a simple but efficient oblivious merge solution that makes the eviction (memory shuffling) cheaper.

5.1 Resizable ORAM

In this section, we introduce *resizability* a new notion that makes recent tree-based ORAM constructions dynamic, and therefore suitable for Cloud scenarios. We introduce different strategies to increase the size of the ORAM while maintaining the same level of security. The main challenge in tree-based ORAM resizability consists of finding the right buckets' size to preserve a similar security level as for static constructions. Some of the proposed strategies inherently depends on the

user's usages habits while some of them clearly outperform naive solution independently. We also present different pruning techniques to also reduce the size of the ORAM tree. We demonstrate in this work that careful analysis is mandatory to understand the relation between the ORAM buckets' size and and the resulting security. Ad-hoc solutions can either lead to an unecessary storage waste or to totally break the ORAM obliviousness.

5.1.1 Motivation and Findings

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

Resizability is a vital property of any ORAM to be used for cloud storage. One of the selling points of cloud services is *elasticity*, the ability to start with a particular footprint and seamlessly scale resources up or down to match demand. Imagine a start-up company that wants to securely store their information in the cloud using ORAM. At launch, they might have only a handful of users, but they expect sometime in the long-term to increase to 10,000. With current solutions, they would have to either pay for the 10,000 users worth of *storage* starting on day one, even though most of it would be empty, or pay for the *communication* to repeatedly reinitialize their database with new sizes as they become more popular. Re-initializing the ORAM would negate any benefit from the new worst-case constructions. Additionally, one can imagine a company that is seasonal in nature (e.g., a tax accounting service) and would like the ability to downsize their storage during off-peak times of the year to save costs.

Consequently, the problem of resizing these new tree-based ORAMs is important for practical adoption in real-world settings. In light of that, we present several techniques for both increasing and decreasing the size of recent tree-based ORAMs to reduce both communication and storage complexity. We focus on constant client memory ORAM (the [129] ORAM) since it is an interest-

ing setting, especially for hardware-constrained devices and large block sizes or situations where multiple parties want to share the same ORAM so need to exchange the state. We are able to show that, although the resizing techniques themselves are intuitive, careful analysis is required to ensure security and integrity of ORAMs. In addition, we show that it is nontrivial to both allow for sub-linear resizing and maintain the constant client memory property of [129] ORAM.

The technical highlights of this work are as follows:

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

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

3. A new, tighter analysis for the [129] ORAM which allows for smaller storage requirements and less communication per query than previous work.

Readers might be interested in the following question: *can we similarly resize poly-logarithmic ORAM constructions?* The answer is not trivial and depends on the underlying construction. For example, for Path ORAM, Ring ORAM [125, 135], a possible solution might be to expand the stash size during the resizing process, however, a careful analysis is required to accurately bound the stash size.

5.1.2 Resizable ORAM

5.1.2.1 Technical Challenges

The challenge behind resizing tree-based ORAMs is threefold:

1. Increasing the size of the tree will have an impact on the bucket size. A leaf node may become an interior node while increasing the ORAM, and vice versa in the decreasing case. The original analysis by [129] differentiates between interior and leaf nodes, while for resizing we will have to generalize the analysis to consider both cases at once.

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

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

5.1.2.2 Resizing Operations

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

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

5.1.3 Adding

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

5.1.3.1 Tightening the bounds

Communication and storage complexities represent the core comparative factor between strategies, and both are dependent primarily on bucket sizes. Consequently, it is important to get a tight

analysis for both interior and leaf bucket sizes. The original bounds for bucket sizes given by [129] are substantially larger than necessary. Therefore, as a first contribution, we give new, tighter bounds for interior and leaf node sizes.

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

$$\Pr(I_i \geq k) = \nu^{-k}. \tag{5.1}$$

In [129], the eviction rate was chosen to be equal to 2 with an overflow probability equal to 2^{-k}, where k here stands for the bucket size. However, if we adjust the bucket size to be $\frac{k}{\log(\nu)}$, the overflow probability is still 2^{-k}, namely $\Pr(I_i \geq \frac{k}{\log(\nu)}) = 2^{-k}$.

This follows from Eq. 5.21 by replacing k by $\frac{k}{\log(\nu)}$. Also, we can investigate the optimal value for the eviction rate ν in terms of communication cost. For $\nu = 4$, we obtain the same overflow probability as with $\nu = 2$ with buckets of half the size. The communication complexity does not change, as we are evicting twice as much, but with buckets of half the size. For larger eviction rates $\nu > 4$ the communication complexity becomes larger. Note that this also reduces the storage by a factor of 2. For N elements stored in the ORAM, the probability that an interior node overflows during eviction computes to

$$\begin{aligned} \Pr(\exists i \in [\nu \cdot \log N] :\ I_i \geq \frac{k}{\log(\nu)}) &= 1 - \Pr(\forall i \in [\nu \cdot \log N] :\ I_i < \frac{k}{\log(\nu)}) && (5.2) \\ &= 1 - \prod_{i=1}^{\nu \cdot \log N} (1 - \Pr(I_i \geq \frac{k}{\log(\nu)})) && (5.3) \\ &= 1 - (1 - 2^{-k})^{\nu \cdot \log N}. \end{aligned}$$

In particular for $\nu = 4$, the optimal choice of the eviction rate,

$$\Pr(\exists i \in [4 \cdot \log N] :\ I_i \geq \frac{k}{2}) = 1 - (1 - 2^{-k})^{4 \cdot \log N}.$$

The buckets that can overflow during an access are limited to those in the paths accessed during the eviction, i.e., $\nu \cdot \log N$ buckets accessed. Also, the number of buckets taken into account is

actually $\nu \cdot \log N$ instead of $2\nu \cdot \log N$. This follows from the fact that for every parent, we write only one real element to one child. Consequently, per eviction and per level, only one child can overflow. For Eq. 5.3, an equality still holds since the buckets can be considered independent in steady state [76].

Given security parameter λ, to compute the size of interior buckets, we solve the equation $2^{-\lambda} = 1 - (1 - 2^{-k})^{\nu \cdot \log N}$ to $k = -\log\left(1 - (1 - 2^{-\lambda})^{\frac{1}{\nu \cdot \log N}}\right)$.

For example, to have an overflow probability equal to 2^{-64}, $\lambda = 64$, $N = 2^{30}$, $\nu = 4$, the bucket size needs to be only 36 while [129] determined the bucket size be equal 72 for the same overflow probability. Moreover, since N, the number of elements in the ORAM, has a logarithmic effect on the overflow probability, the size of interior nodes will not change for large fluctuations of the number of elements N. For example, for $N = 2^{80}$, the interior node still has size 36 with overflow probability 2^{-64}.

Leaf Nodes Let B_i denote the random variable describing the size of the ith leaf node. Thinking of a leaf node as a bin, a standard balls and bins game argument provides us the following upper bound

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

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

$$\begin{aligned} \Pr(\exists i \in [N] : \ B_i \geq k) &= \Pr(\bigcup_{i=1}^{N} B_i \geq k) \\ &\leq \sum_{i=1}^{N} \Pr(B_i \geq k) \\ &\leq \frac{N}{e^{k \cdot (\ln(k) - 1)}}. \end{aligned} \tag{5.4}$$

Note that in Eq. 5.4, we have used the union bound. Based on the same parameters as in the previous example, the size of a leaf node has to be set only to 28 to have an overflow probability equal to 2^{-64}. To compute this result, one solves the equation $k = e^{W(\frac{\log 2^{\lambda} \cdot N}{e}) + 1}$, where $W(.)$ is the product log function. While the size of the interior node can be considered constant for large

fluctuations of N, the size of a leaf node should be carefully chosen depending on N. Every time the number of elements increases by a multiplicative factor of 32, we have to increase the size of the leaf node by 1 to keep the same overflow probability.

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

5.1.3.2 1^{st} Strategy: naive expansion

Let N and n respectively denote the number of leaf nodes and elements in the ORAM. The naive solution is simply adding a new leaf level, as soon as the condition $n > N$ occurs. The main drawback of this first naive solution is the waste of storage which can be explained from two different perspectives. The first storage waste consists of creating, in average, more leaf nodes than elements in the ORAM. The second storage waste in the under-usage of the leaf nodes while they can hold more elements with a slight size increase. Our second strategy will try to get rid of this drawback.

5.1.3.3 2^{nd} Strategy: lazy expansion

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

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

To be able to store more elements, our idea is to slightly increase the leaf bucket size. Therewith, we can keep the same overflow probability. Note the tradeoff between increasing the size of leaf nodes and the communication complexity of the ORAM. To read or write an element in the ORAM, the client downloads the path starting from the root to the leaf node. If the size of this path

(when increasing the size of the bucket) is larger than a regular ORAM tree with the same number of elements, then this technique would not be worth applying.

[65] have shown that by increasing the leaf node size from k to $\alpha + k$, we can reduce the storage overhead while handling more elements than leaf nodes. For N leaf nodes, we can have up to $\alpha \cdot N$ elements. While [65] chose α to optimize the storage cost for a given overflow probability, we instead target the computation of the value α for the optimal communication complexity. In our subsequent analysis, the previous bounds for interior and leaf node sizes as computed in the previous section are used.

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

Lemma 5.1.1. *Let x denote the optimal leaf bucket size for the 2^{nd} strategy. Then,*

$$\alpha = \frac{x}{e} \cdot (\frac{2^{-\lambda}}{N})^{\frac{1}{x}} \tag{5.5}$$

holds, where λ is the security parameter and N the number of leaf nodes.

Proof. By a balls-and-bins argument, we are in a scenario where we insert uniformly at random $\alpha \cdot N$ balls into N bins. The ith bin overflows if there are x balls from $\alpha \cdot N$ that went to the same ith bin. The possible number of combinations equals $\binom{\alpha \cdot N}{x}$. By applying the upper bound inequality to the probability of the union of events (possible combinations), we obtain

$$\begin{aligned} \Pr(B_i \geq x) &\leq \binom{\alpha \cdot N}{x} \cdot \frac{1}{N^x} \\ &\leq (\frac{e \cdot \alpha \cdot N}{x})^x \cdot \frac{1}{N^x} \\ &= (\frac{e \cdot \alpha}{x})^x. \end{aligned}$$

Computing the union bound over all leaf nodes results in

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

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

■

Corollary 5.1.1. *Let k denote the size of the interior node. The best communication complexity for the 2^{nd} strategy is achieved iff the leaf bucket size x equals*

$$x = \frac{\frac{k}{\ln 2} + \sqrt{k - 4 \cdot k \cdot \log \frac{2^{-\lambda}}{N}}}{2}$$

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

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

The best value of x for a fixed value of y, k and λ is the maximum value of the function f defined as

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

The first derivative of f is $\frac{df}{dx}(x) = x^2 - \frac{k}{\ln(2)} \cdot x + k \cdot \log \frac{2^{-\lambda}}{N}$. This quadratic equation has only one valid solution for a non-negative leaf buckets size and $2^{\lambda} >> N$. The only valid root for the first derivative is $x = \frac{\frac{k}{\ln 2} + \sqrt{k - 4 \cdot k \cdot \log \frac{2^{-\lambda}}{N}}}{2}$.

■

Once we have computed the optimal leaf node size, we can plug the result into Eq. 5.5 to compute the optimal value α. For example, for $N = 2^{30}$ leaves, the size of the leaf bucket in the naive strategy is $y = 28$, the size of the interior node $k = 36$. Applying the result of Corollary 5.1.1 outputs the size of the leaf bucket for an optimal communication complexity which is equal to

$x \approx 85$. Applying the result of Lemma 5.1.1, we obtain $\alpha \approx 15$. The communication complexity saving compared to the naive strategy is around 7% while the storage savings is a significant 87%.

One disadvantage of the 2^{nd} strategy is the possibility of storage underutilization. Imagine the client stores $\alpha \cdot N$ elements in the ORAM tree. When adding a new element, it will trigger the creation of a new leaf level, which is a waste of storage. For example, the client can have $\alpha \cdot N + 1$ elements in his ORAM tree, then performs a loop which respectively adds and deletes two elements. This loop will imply the allocation of an unused large amount of storage (in $O(N)$). Also, this loop implies leaf node pruning which is more expensive (in term of communication complexity) compared to leaf increasing as we will see in Section 5.1.4.

5.1.3.4 3^{rd} strategy: dynamic expansion

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

Let us assume that we start with a full binary tree containing $N = 2^{\text{l}}$ leaf nodes. Dynamic insertion results in the creation of two different levels of leaves. The first one is on the lth level while the other one in on the $(l+1)$th level. In general, after adding $\eta \cdot \alpha$ elements, the number of leaves in the lth level is equal to $N - \eta$ while the number of leaves in the $(l+1)$th level is equal to 2η.

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

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

Lemma 5.1.2. *Let B_i denote the random variable describing the size of the ith leaf node, $1 \leq i \leq N+\eta$. For the 3^{rd} strategy and a bucket of size B_i, the overflow probability computes to*

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

Proof. After adding η leaf nodes to the structure, the ORAM contains $N+\eta$ leaves. The probability that at least one leaf node has size larger than k is

$$\begin{aligned} \Pr(\exists i \in [N+\eta] : \ B_i \geq k) &= \Pr(\bigcup_{i=1}^{N+\eta} B_i \geq k) \\ &\leq \sum_{i=1}^{2\cdot\eta} \Pr(B_i \geq k) + \sum_{i=2\cdot\eta+1}^{N+\eta} \Pr(B_i \geq k) \end{aligned} \tag{5.6}$$

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

$$\begin{aligned} \text{for } 1 \leq i \leq 2 \cdot \eta &: \ \Pr(B_i \geq k) \leq \binom{\alpha \cdot (N+\eta)}{k} \cdot (\frac{1}{2 \cdot N})^{\text{k}} \\ \text{for } 2 \cdot \eta + 1 \leq i \leq N + \eta &: \ \Pr(B_i \geq k) \leq \binom{\alpha \cdot (N+\eta)}{k} \cdot (\frac{1}{N})^{\text{k}}. \end{aligned}$$

Note that $\alpha \cdot (N+\eta)$ is the current number of elements in the ORAM. We plug both inequalities in to Eq. 5.6 and get

$$\begin{aligned} \Pr(\exists i \in [N+\eta] : \ B_i \geq k) &\leq 2 \cdot \eta \cdot \binom{\alpha \cdot (N+\eta)}{k} \cdot (\frac{1}{2 \cdot N})^{\text{k}} + (N-\eta) \cdot \binom{\alpha \cdot (N+\eta)}{k} \cdot (\frac{1}{N})^{\text{k}} \\ &\leq (\frac{2 \cdot \eta}{2^k} + N - \eta) \cdot (1 + \frac{\eta}{N})^k \cdot (\frac{e \cdot \alpha}{k})^k. \end{aligned}$$

The bound above is depending on η. Thus, we now compute the value of $\eta < N$ maximizing the bound. This leads us to the function $g(\eta) = (\frac{2 \cdot \eta}{2^k} + N - \eta) \cdot (1 + \frac{\eta}{N})^k$. Function g has a local

maximum value for any $\eta, 1 \leq \eta \leq N$ such that $\eta_{max} = \frac{N}{A} \cdot \frac{k-A}{A(k+1)}$ where $A = 1 - \frac{1}{2^{k-1}}$. We replace η_{max} in g to get an upper bound for any any η and $k \geq 2$,

$$\begin{aligned}
\Pr(\exists i \in [N+n] : \ B_i \geq k) &\leq g(n_{max}) \cdot (\frac{e \cdot \alpha}{k})^k \\
&\leq N \cdot \frac{A+1}{k+1} \cdot (\frac{k(A+1)}{A(k+1)})^k \cdot (\frac{e \cdot \alpha}{k})^k \\
&\leq \frac{2 \cdot N}{k+1} \cdot (\frac{2 \cdot e \cdot \alpha}{k})^k.
\end{aligned}$$

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

■

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

5.1.3.5 Comparison of Strategies

We present a comparison between our three strategies in terms of storage complexity (Figure 5.2) and communication complexity per access (Figure 5.8). We perform our comparison on a block level, thereby remaining independent of the actual block size.

Communication complexity: the 2^{nd} strategy offers best communication complexity. This is due to shorter paths, a result of flatter trees – compared to the naive 1^{st} solution. Also, compared to the 3^{rd} strategy, the leaf buckets have smaller size. For a number of elements $N = 2^{30}$ and 2^{-64} overflow probability, the interior node size equals 36 which is appropriate for all three strategies. The difference consists of the size of the leaf buckets as well as the height of the resulting tree. The bucket size for the naive (1^{st}), lazy (2^{nd}) and dynamic (3^{rd}) strategy respectively equals 28, 85 and 130 blocks. The tree's height for the naive solution equals 30 while for the lazy and dynamic solution the tree height is 26 since $\alpha \approx 2^4$. In Figure 5.8, for an eviction rate used equals 4, the entire communication complexity (upload/download) on the main ORAM respectively equals 26928, 24210 and 25020 blocks for the naive, lazy and dynamic solution. Note that per access,

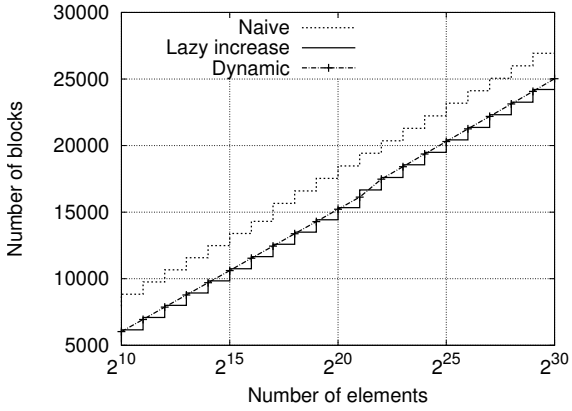

Figure 5.1: Communication, blocks per access

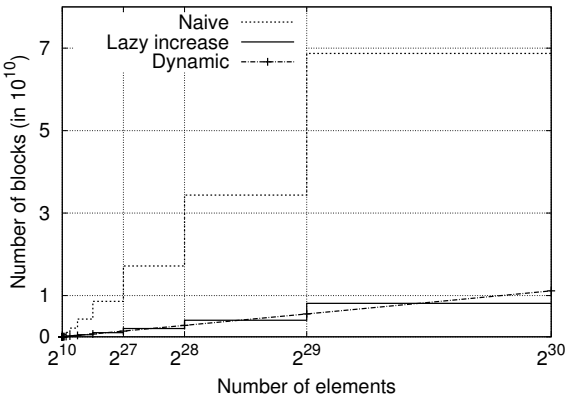

Figure 5.2: Storage cost, blocks

we save around 7% in communication cost. Recall that our main purpose is to reduce the storage overhead while maintaining the same communication complexity. However, our results show that storage optimization has a direct consequence on reducing the communication complexity as well.

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

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

In general, both the 2^{nd} and 3^{rd} strategies outperform the naive one in terms of communication and storage complexities.

5.1.3.6 Position Map

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

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

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

(2) The second solution is straightforward and based on creating a new level of recursion when a new level of leaves is created. Note that blocks in this level will have $O(\tau \cdot \log N + 1)$ bits instead $O(\tau \cdot \log N)$. To access an "old" element, we use the same method described above.

5.1.4 Pruning

Assume an ORAM storing N elements. Now, the client deletes η elements from the ORAM. Consequently, the naive ORAM construction now contains $N - \eta$ elements, but still has N leaves.

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

5.1.4.1 Lazy pruning

In Section 5.1.3.3, we have demonstrated that leaves can store significantly more elements while only slightly increasing their size. We will use this observation to construct a new algorithm for lazy pruning. Assume that the leaf level contains N leaves for $\alpha \cdot N$ elements stored. Let η denote the number of elements deleted by the client. For sake of simplicity, assume that, at the beginning, we have $\eta = 0$ and N leaf nodes. Our pruning technique is similar to the "lazy" insertion described previously. Whenever $\alpha \cdot \frac{N}{2} < \eta \leq \alpha \cdot N$, we keep the same number of leaves. Within this interval, the client can add or delete elements without applying any change to the structure, as long as the number of elements remains within the defined interval. If the number of deletion equals $\alpha \cdot \frac{N}{2}$, the client proceeds to remove an entire level of leaf nodes. The client proceeds to read every leaf node, along with its sibling, and merges them with their parent node. While this appears to be straightforward, an oblivious merging of siblings into their parent is more complex under our constant-client memory constraint. We will discuss this in great detail below.

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

5.1.4.2 Dynamic pruning

Given that pruning an entire level at once is very inefficient, we now investigate how pruning can be done in a more gradual way. For every α elements we delete, we will prune two children

and merge their contents into their parent node. The pruning will *fail* if the number of elements in both children and parent is more than k. This can only occur if there are more than k elements associated (tagged) to these children. The following lemma states the upper bound of the overflow probability for the parent node after a merging. Recall that we begin with a full binary tree of N leaves and $\alpha \cdot N$ elements. Assume that we have already deleted $\alpha(\eta - 1)$ elements, and we want to delete an additional α elements.

Lemma 5.1.3. *Let P_η denote the random variable of the size of the ηth parent node. For dynamic pruning, the probability that pruning will fail equals*

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

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

$$\begin{aligned}
\Pr(P_\eta > k) &= \binom{\alpha \cdot (N - \eta)}{k} \cdot (\frac{2}{N})^k \\
&\leq (\frac{e \cdot \alpha \cdot (N - \eta)}{k})^k \cdot (\frac{2}{N})^k \\
&\leq (1 - \frac{\eta}{N})^k \cdot (\frac{2e \cdot \alpha}{k})^k \\
&\leq (\frac{2e \cdot \alpha}{k})^k
\end{aligned}$$

■

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

Complexity of oblivious merging The cost of dynamic pruning boils down to the cost of obliviously merging three buckets of size k. We can achieve this with $O(k)$ communication and constant memory complexity. First, note that we do not have to merge all three buckets at once. All that

Algorithm 2: GeneratePermutation(A, B)

Input: Configuration of buckets A and B
Output: A permutation which randomly "lines up" bucket B to bucket A
// Slots in A and B start either empty or full; mark slots in A as ''assigned'' if block from B is assigned in π
$x \leftarrow$ number of empty slots in A ;
$y \leftarrow$ number of full slots in B ;
$d \leftarrow x - y$;
for i *from 1 to* k **do**
  if $B[i]$ *is full* **then**
    $z \xleftarrow{\$}$ all empty slots in A;
  else
    if $d > 0$ **then**
      $z \xleftarrow{\$}$ all non-assigned slots in A;
      $d \leftarrow d - 1$;
    else
      $z \xleftarrow{\$}$ all full slots in A;
    end
  end
  $\pi[i] \leftarrow z$;
  $A[z] \leftarrow$ assigned ;
end
return π ;

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

In Algorithm 2, the client randomly permutes the order of the elements in one bucket, subject to the constraint that, for all indices, at most one of the elements between both buckets is real. That is, the permutation "lines up" the two buckets so that they can be merged efficiently. Special care must be given to generate this permutation using only constant memory. The client makes use of "configuration maps" which simply indicate, for every slot in a bucket, whether that slot is currently full or empty. These maps can be stored encrypted on the server and take up $O(1)$ space each in terms of blocks (because the buckets contain $O(\log N)$ elements and a single block is at least $\log N$ bits [129, 135]). Then, the client iterates through the slots in one bucket, randomly

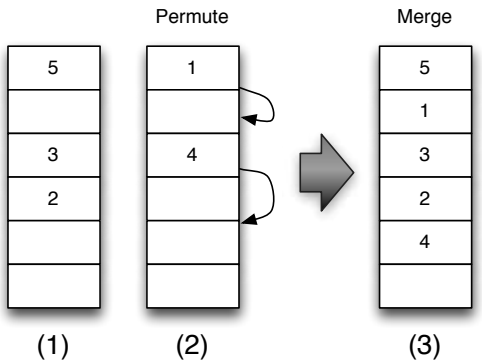

Figure 5.3: Illustration of permute-and-merge process. Bucket (2) is permuted and then merged with bucket (1) to create a new, combined bucket (3).

pairing them with compatible slots in the other (i.e., a full slot cannot be lined up with another full slot). An additional twist is that an empty slot can be lined up with either a full or empty slot in the other bucket, but not at the expense of "using up" an empty slot that might be needed later since we cannot match full with full. Therefore, we have to also keep a counter of the difference between empty slots in the target bucket and full slots in the source bucket.

As seen in Figure 5.3, once the client generates the permutation, they can retrieves the elements pairwise from both buckets (i.e., slot i from one bucket and the slot which is mapped to i via the permutation from the other bucket), writing back the single real one to the merged bucket.

It remains to show that this permutation does not reveal any information to the adversary. If it was a completely random permutation, it would certainly contain no information. However, we are choosing from a reduced set: all permutations which cause the bucket to "line up" with its sibling.

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

To make this approach work, we need to slightly modify the behavior of the bucket ORAMs. Previously, when a new element was added to a bucket, it did not matter which slot it went into in that bucket. It was possible, for instance, that all the real elements would be kept at the top of

the bucket and, when adding a new one, the client would simply insert that element into the first empty slot that it could find. However, to use this permutation method we require that the buckets be in a random "configuration" in terms of which slots are empty and which are filled. Therefore, when inserting an element, the client should choose randomly amongst the free slots. Again, this is possible with constant client memory using our configuration maps. With this behavior, applying the above logic leads to the conclusion that the adversary learns nothing about the load of the bucket from seeing the permutation.

Note. Following the result by Nayak et al. [115], for security reasons, two buckets cannot be obliviously merged if one bucket at least hasn't been shuffled after a previous merge. In order to avoid this constraint, the client can reshuffle any bucket using PIR write vectors, similar to Onion ORAM. This maintains a constant client memory overhead while eliminating the above constraint.

Section Conclusion. We have shown in this section how to resize constant client memory tree-based ORAM [129] to fit users' storage needs. We have demonstrated that resizability can decrease in some settings the monetary cost by an order of magnitude. We also showed that storage and communication are dependent, and saving on one dimension can impact the other. However, it remains an open problem to carefully study the resizability of poly-logarithmic client memory construction.

5.2 Recursive ORAM

There are two major methods for building ORAM schemes: hierarchical or tree-based. Throughout these recent years, many constructions were mainly improving the ORAM eviction techniques to lower the communication overhead while, in most cases, employing one of the two data structures [1]. In this section, we investigate whether there is a better data structure that can achieve better asymptotics while preserving ORAM obliviousness. We demonstrate that binary trees can be generalized to a new recursive tree data structure that greatly reduces the communication and

[1] Recently Fletcher et al. [62] propose a construction that merge both Goldreich and Ostrovsky construction [70] and tree-based ORAMs [135] to get better asymptotics while reducing round complexity.

storage overhead. We also show that most of the tree-based ORAM schemes can be easily plugged in this new data structure with *no* impact on the scheme's functionalities. We theoretically and empirically show that our constructions perform better than all previous known constructions. We selected recent tree-based constructions such as Path ORAM [135], Shi et al. [129] and Gentry et al. [65] an we replace their data structures by our recursive trees. We provide our technical findings in the subsequent section.

5.2.1 Contribution Summary

We introduce a novel recursive tree-based ORAM. We show that traditional binary (or κ-ary) tree structure in tree-based ORAM does not represent the optimal structure to use in an ORAM. We introduce a new recursive structure that further reduces the ORAM asymptotics. Our new data structure reduces the *average* or *expected* path length, therefore reducing the cost to access blocks. Our goal is to support both constant and poly-log client memory ORAMs. Straightforward techniques to reduce the tree height, e.g., by using κ-ary trees [65], require poly-logarithmic client memory due to the more complex eviction mechanism. The idea behind our technique called r-ORAM is to store blocks in a recursive tree structure. The proposed recursive data structure substitutes traditional κ-ary ($\kappa \geq 2$) trees with better communication. Starting from an *outer* tree, each node in a tree is a root of another tree. After r trees, the recursion stops in a *leaf* tree. The worst-case path length of r-ORAM is equal to $c \cdot \log N$, with $c = 0.78$, yet this worst-case situation occurs only rarely. Instead in practice, the *expected* path length for the majority of operations is $c \cdot \log N$, with $c = 0.65$ for binary trees. The shortest paths in binary trees have length $0.4 \cdot \log N$. In addition to saving on communication, the r-ORAM approach also saves up to 0.8 on storage due to fewer nodes in the recursive trees. To support our theoretical claims, we have also implemented r-ORAM and evaluated its performance. The source code is available for download [18].

r-ORAM is a *general* technique that can be used as a building block to improve any recent tree-based ORAM, both with $O(1)$ client memory such as [129], $O(\log N)$ client memory such as [135], and $O(\log^2 N)$ client memory such as [65] – and variations of these ORAMs. In addition to binary tree ORAM, r-ORAM can also be applied to κ-ary trees. Targeting practicality, we abide

from non-tree based poly-log ORAMs, such as [90]. While they achieve $O(\frac{\log^2 N}{\log \log N})$ worst-cast communication cost, their approach induces a large constant ~ 30.

5.2.2 Recursive Binary Trees

A Naive Approach: To motivate the rationale behind r-ORAM, we start by describing a straightforward attempt to reduce the path length and therewith communication cost. Currently, data elements added to an ORAM are inserted to a tree's root and then percolate down towards a randomly chosen leaf. As a consequence, whenever a client needs to read an element, the whole path from the tree's root to a specific leaf needs to be downloaded. This results in path lengths of $\log N$.

A naive idea to reduce path lengths would be to percolate elements to any node in the tree, not only leaves, but also interior nodes. To cope with added elements destined to interior nodes, the size of nodes, i.e., the number of elements that can be stored in such *buckets*, would need to be increased. At first glance, this reduces the path length. For example, the minimum path length now becomes 1. However, the distribution of path lengths with this approach is biased to its maximum length of $\log N$: for a tree of N nodes, roughly $\frac{N}{2}$ are at the leaf level. Thus, the expected path length would be $\approx \log(N) - 1$, resulting in negligible savings. This raises the question whether a better technique exists, where the distribution of path lengths can be "adjusted".

r-ORAM Overview: We first give an overview about the structure of our new recursive ORAM constructions. In r-ORAM, parameter r denotes the recursion factor. Informally, an r-ORAM comprises a single *outer* binary tree, where each node (besides the root) is the root of an *inner* binary tree. Recursively, a node in an inner tree is the root of another inner tree, cf. Figure 5.4. After the outer tree and $r - 1$ inner trees, the recursion ends in a binary *leaf* tree. That is, each node (besides the root) in an $(r - 1)$th inner tree is the root of a leaf tree. The fact that a root of a tree is never a (recursive) root of another tree simply avoids infinite duplicate trees.

Let the outer tree have y leaves and height $\log y$, where y is a power of two and $\log$ the logarithm base 2. Also, inner trees have y leaves and height $\log y$. Leaf trees have x leaves, respectively, and height $\log x$. The number of elements N that can be stored in an r-ORAM equals the total number of leaves in all leaf trees, similarly to related work on tree-based ORAM [129].

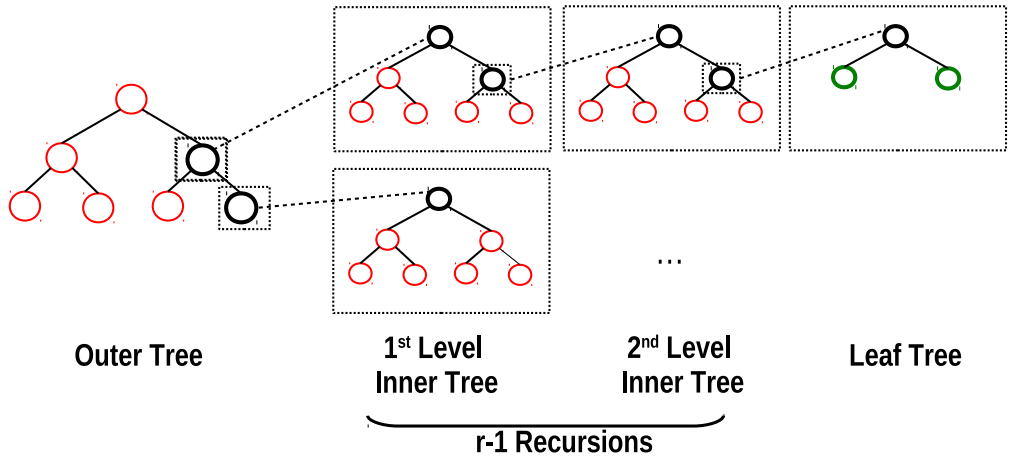

Figure 5.4: Structure of an r-ORAM

5.2.2.1 Storage Cost

For a total number of N elements, we have N corresponding leaves in r-ORAM. To compute the total number of nodes ν, we start by counting the number of leaf trees in r-ORAM. For the outer tree, we have $2y-2$ possible nodes which are the root for another recursive inner tree. Each inner tree has also $2y-2$ nodes, and since we have $r-1$ levels of recursion aside from the outer tree, the following equality holds:

$$N = (2y-2)\cdot(2y-2)^{r-1}\cdot x = (2y-2)^r\cdot x \tag{5.7}$$

$$= 2^r\cdot x\cdot(y-1)^r. \tag{5.8}$$

Each of the nodes in an r-ORAM is a *bucket* ORAM of size z, where z is a security parameter, e.g., $z = O(\log N)$ [129]. The total number of nodes ν, with N leaves, in an r-ORAM (main tree) is the sum of all nodes of all leaf trees plus the nodes of all inner trees, the outer tree, and its root, i.e.,

$$\begin{aligned}\nu(N) &= (2y-2)^r\cdot(2x-2)+\sum_{i=0}^{r}(2y-2)^i \\ &\overset{(5.7)}{=} (2N-2\cdot\frac{N}{x})+\frac{(2y-2)^{r+1}-1}{(2y-2)-1} \\ &= 2N+(\frac{2y-2}{2y-3}-2)\cdot\frac{N}{x}-\frac{1}{2y-3}.\end{aligned}$$

Thus, the total storage cost for r-ORAM is $\nu(N)\cdot z\cdot l$ with blocks (bucket entries) of size l bits. This storage does not take into account the position map. The total storage of the entire r-ORAM

structure equals $\nu(N) \cdot z \cdot l + \sum_{i=1}^{\lceil \frac{\log N}{\log \beta} \rceil - 1} z \cdot \nu(\frac{N}{\beta^i}) \cdot \log \frac{N}{\beta^{i-1}}$, where β is the position map factor. For $l = \omega(\log^2 N)$ the sum in the storage complexity is negligible. The total storage then equals $\nu(N) \cdot z \cdot l$.

For appropriate choices of x and y, discussed in the next section, r-ORAM reduces the storage cost in comparison with the $(2N - 1) \cdot z \cdot l$ bits of storage of related work. So for example, with $x = 2$ and $y = 4$, the storage is equal to $\frac{8N}{5}$ resulting in a reduction by 20% of the number of nodes compared to existing tree-based ORAMs. However, this does not mean the same reduction for storage overhead. In fact, Section 5.2.4 will show that the size of the bucket can be reduced for [129]'s ORAM and increased for Path ORAM. Consequently, our storage saving varies between 4% to 20% depending on the ORAM.

As of Eq. (5.8), for a given number of elements N, r-ORAM depends on three parameters: recursion factor r, the number of leaves of an inner/outer tree y, and the number of leaves of a leaf tree x. We will now describe how these parameters must be chosen to achieve maximum communication savings.

5.2.2.2 Communication Cost

In ORAM, the "communication cost" is the number of bits transferred between client and server. We now determine the communication cost of reading an element in r-ORAM, e.g., during a *ReadAndRemove* operation. Reading an element implies reading the entire path of nodes, each comprising of z entries, and each entry of size l bits. In related work, any element requires the client to read a ***fixed*** number of $\log N \cdot l \cdot z$ bits. For the sake of clarity in the text below, we only compute the number of nodes read by the client, i.e., without multiplying by the number of entries z and the size of each entry l. Since the main data tree and the position map have different block sizes, computing the height of r-ORAM independently of the block size enable us to tackle both cases at the same time. At the end, to compute the exact communication complexity of any access we can just multiply the height with the appropriate block sizes, see Section 5.2.2.5.

A path going over a node on the ith level in the outer tree requires reading one bucket ORAM less than a path going over a node on the $(i + 1)$th level in the outer tree. Consequently with

r-ORAM, we need to analyze its best-case communication cost (shortest path), worst-case cost (longest path), and most importantly the average-case cost (average length).

The worst-case cost to read an element in r-ORAM occurs when the path comprises nodes of the full height of every inner tree until its leaf level, before finally reading the corresponding leaf tree. The worst-case cost $\mathcal{C}$ equals

$$\mathcal{C}(r, x, y) = r \cdot \log y + \log x. \tag{5.9}$$

The best-case occurs when the path comprises one node of every inner tree before reading the leaf tree. The best-case cost $\mathcal{B}$ equals

$$\mathcal{B} = r + \log x. \tag{5.10}$$

The worst-case cost in this setting is a function of three parameters that must be carefully chosen to minimize worst- and best-case cost. Theorem 5.2.1 summarizes how the recursion factor r, the number of leaves y in inner trees, and the number of leaves in leaf trees x have to be selected.

Minimizing the worst-case path length is crucially important, as it also determines the average path-length. We will see later that the distribution of paths' lengths (and therewith the cost) follows a normal distribution. That is, minimizing the worst case also leads to a minimal expected case and therewith the best configuration for r-ORAM. Similarly, as the paths' lengths follow a normal distribution, average and median cost are equivalent.

A client can use the minimal worst-case parameters to achieve the "cheapest configuration" for a r-ORAM structure storing a given number of elements N.

Theorem 5.2.1. *If* $r = \log((\frac{N}{2})^{\frac{1}{2.7}})$, $x = 2$, *and* $y = \frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1$, *the worst-case cost* $\mathcal{C}$ *is minimized and equals*

$$\mathcal{C} = 1 + 2.08 \cdot \log((\frac{N}{2})^{\frac{1}{2.7}}) \approx 0.78 \cdot \log N.$$

The best-case cost $\mathcal{B}$ *is* $\mathcal{B} = 1 + \log((\frac{N}{2})^{\frac{1}{2.7}}) \approx 0.4 \cdot \log N$.

Proof. Function $\mathcal{C}$ depends on three variables that we can reduce to two by substituting Eq. (5.21) into Eq. (5.22). From Eq. (5.21), we have $\log x = \log(N) - r - r \cdot \log(y - 1)$. The worst-case cost

then computes to

$$\mathcal{C}(r, y) = \log(N) - r + r \cdot \log(\frac{y}{y-1}). \tag{5.11}$$

By fixing $r > 0$, the worst-case cost is a non-increasing function in y, since $y \mapsto \log(\frac{y}{y-1})$ is a non-increasing function for $y > 1$. Thus, for any non-negative r, the minimum value of the worst cost is smaller for larger values of y. Also, with $x \geq 2$, the number of the leaves of inner trees y is upper bounded: $N \geq 2 \cdot (2y-2)^r \Rightarrow y \leq \frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1$. For small x, we therewith get a larger upper bound for y. Therefore, we have to fix x to its minimum value which equals 2. This could not be inferred from Eq. 5.22 while not decreasing the number of variables of the linear system. The optimum number of leaves for the inner trees then equals $y = \frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1$. Putting these values back in Eq. (5.23), results in $\mathcal{C}$ depending on only one variable r, the recursion factor:

$$\mathcal{C}(r) = 1 + r \cdot \log(\frac{1}{2}.(\frac{N}{2})^{\frac{1}{r}} + 1) \tag{5.12}$$

Finally, we derive the minimum of the worst-case cost by computing the first derivative of the convex function $\mathcal{C}(r)$. The derivative is $\frac{d\mathcal{C}}{dr}(r) = \log(\frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1) - \frac{\ln(\frac{N}{2}) \cdot (\frac{N}{2})^{\frac{1}{r}}}{2r \cdot (\frac{1}{2} \cdot (\frac{N}{2})^{\frac{1}{r}} + 1)}$.

We achieve $\frac{d\mathcal{C}}{dr}(r) \approx 0$ for $r' \approx \log((\frac{N}{2})^{\frac{1}{2.7}})$. Since $\mathcal{C}(r)$ is convex, the value of r' is the minimum for any $r \leq \log(N) - 1$. Replacing r' in equations (5.10) and (5.12) gives the worst-case and best-case costs of the theorem, therefore completing the proof.

■

Careful readers will notice that we have bounded x to be at least equal to 2 in our theorem's proof. If we consider that $x = 1$, we do not therefore have any leaf tree at the end. Thus, there are some nodes in the last recursion that will behave as leaves and interior buckets at the same time. This will have some critical issues in term of security proofs. Considering a node as interior and leaf node at the same time will basically double the size (because in the analysis we have to consider the disjunction of both events). Fixing $x = 1$, seems a good idea that simplifies greatly the analysis, however it will not help in optimizing the communication overhead. In fact, the size, in bits, to download a path will be equal to the same path with $x = 2$ with two times more elements.

5.2.2.3 Average-Case Cost

While the parameters for a minimal worst-case cost also lead to a minimal average-case cost, we still have to compute the average-case cost. The cost of reading an element ranges from $\mathcal{B}$, the best-case cost, to $\mathcal{C}$, the worst-case cost. Also, due to the recursive structure of the r-ORAM, the average-case cost of accessing a path is not uniformly distributed.

In order to determine the average-case cost, we count, for each path length i, the number of leaves that can be reached. That is, we compute the *distribution* of leaves in an r-ORAM with respect to their path length starting from the root of the outer tree. Let non-negative integer $i \in (\mathcal{B}, \mathcal{B}+1, \ldots, \mathcal{C})$ be the path length and therewith communication cost. We compute $\mathcal{N}(i)$, the number of leaves in a leaf tree that can be reached by a path of length i. Thus, the average cost, $\mathcal{A}_v$ can be written as $\mathcal{A}_v = \frac{\sum_{i=\mathcal{B}}^{\mathcal{C}} i \cdot \mathcal{N}(i)}{N}$, where N is the total number of elements and therefore leaves in the r-ORAM.

Theorem 5.2.2. *For:*

$$\mathcal{N}(i) = 2^i \cdot \sum_{j=0}^{r} (-1)^j \binom{r}{j} \binom{i - \log(x) - j \cdot \log(y) - 1}{r-1},$$

the average cost of a r-ORAM access is $\mathcal{A}_v = \frac{\sum_{i=\mathcal{B}}^{\mathcal{C}} i \cdot \mathcal{N}(i)}{N}$.

Proof. Counting the number of leaves for a path of length i is equivalent to counting the number of different paths of length i. The intuition behind our proof below is that the number of different paths of length i can be computed by the number of different paths in the r recursive trees $\mathcal{R}(i)$ times the number of different paths in the leaf tree, $\mathcal{N}(i) = \mathcal{R}(i) \cdot \mathcal{W}(i)$.

As stated earlier, the leaf tree has x leaves, $\mathcal{W}(i) = 2^{\log x} = x$.

To compute $\mathcal{R}(i)$, we introduce an array A_r of r elements. For a path $\mathcal{P}$ of length i, element $A_r[j], 1 \leq j \leq r$, stores the number of nodes in the jth inner tree that have to be read, i.e., the maximum level in the jth tree that $\mathcal{P}$ covers. For a path $\mathcal{P}$ of length i, we have $i = \sum_{j=1}^{r} A_r[j] + \log(x)$. For all j, $1 \leq A_r[j] \leq \log(y)$. For any path $\mathcal{P}$ of length i, we can generate $2^{i-\log(x)}$ other possible paths covering exactly the same number of nodes in every recursive inner tree, but taking

different routes on each of them. For illustration, let path $\mathcal{P}$ go through two levels in the second inner tree – this means that there are actually 2^2 other paths that go through the same number of nodes. Therefore, if we denote the possible number of *original* paths of length i by $\mathcal{K}(i)$, the *total* number of paths equals $\mathcal{R}(i) = 2^{i-\log(x)} \cdot \mathcal{K}(i)$, for any integer $i \in \{\mathcal{B}, \ldots, \mathcal{C}\}$. We compute $\mathcal{K}(i)$, by computing the number of solutions of equation

$$A_r[1] + A_r[2] + \cdots + A_r[r] = i - \log x$$

$$\Leftrightarrow$$

$$(A_r[1] - 1) + \cdots + (A_r[r] - 1) = i - r - \log x. \tag{5.13}$$

Computing the number of solutions of Eq. (5.13) is equivalent to counting the number of solutions of packing $i - r - \log x$ (indistinguishable) balls in r (distinguishable) bins, where each bin has a finite capacity equal to $\log(y) - 1$. Here, $A_r[j] - 1$ denotes the size of the bin. This can be counted using the stars-and-bars method leading to $\mathcal{K}(i) = \sum_{j=0}^{r}(-1)^j \binom{r}{j}\binom{i-\log(x)-j\cdot\log(y)-1}{r-1}$. With $\mathcal{N}(i) = 2^i \cdot \mathcal{K}(i)$, we conclude our proof.

■

The average as formalized in the previous theorem does not give any intuition about the behavior of the average cost. For illustration, we plot the exact combinatorial behavior of the distribution of the leaf nodes. We present two cases that show the behavior of the leaf density, i.e., the probability to access a leaf in a given level in r-ORAM. We compute as well the average cost of accessing r-ORAM in two different cases, for $N = 2^{32}$ and $N = 2^{42}$, see Figure 5.5.

We can simplify our average-case equation. The number of possibilities $\mathcal{K}$ of indistinguishable balls packing in distinguishable bins can be approximated by a normal distribution [33, 35]. For a given level $i \in \{\mathcal{B}, \cdots, \mathcal{C}\}$ we have

$$\mathcal{K}(i) \approx \frac{A}{s\sqrt{2\pi}} \cdot e^{-\frac{(i-r-\log(x)-\frac{c}{2})^2}{2s^2}}, \tag{5.14}$$

where $c = r \cdot (\log(y) - 1)$, $s = \frac{\frac{c}{2}+1}{\varpi}$, $A = r \cdot \log(y)$, and ϖ being the solution of the equation $\varpi \cdot e^{-\frac{\varpi^2}{2}} = \frac{\sqrt{2\pi}\cdot(\frac{c}{2}+1)}{A}$.

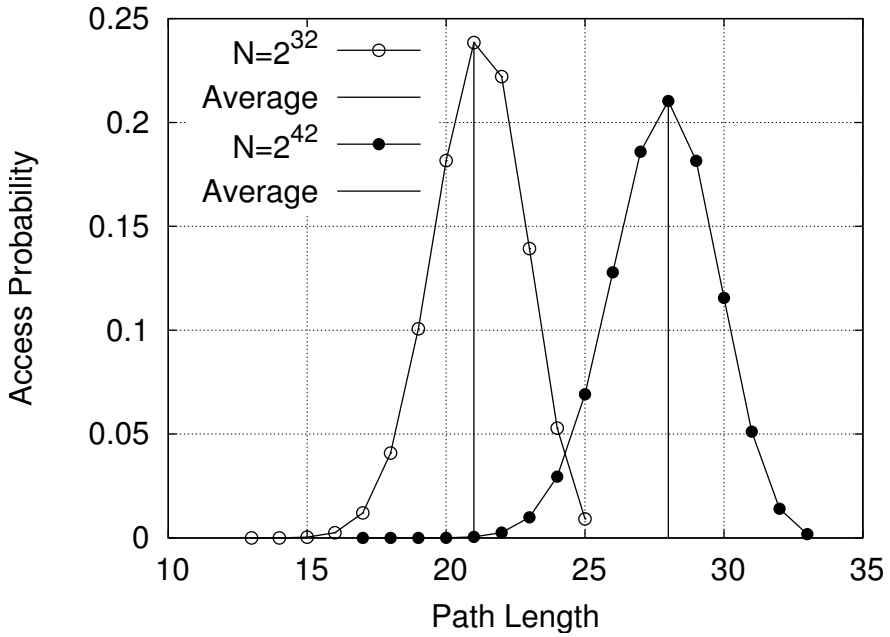

Figure 5.5: r-ORAM path length distribution

Since the number of leaves in the ith level of r-ORAM (over 2^i) follows a normal distribution with a mean $\frac{c}{2}$, which roughly equals the worst case over 2. The average case is the mean of the Gaussian distribution, therefore minimizing the worst case is equivalent to minimizing the average case. Thus, we can use the same parameters obtained in Th. 5.2.1 to compute the minimal value of the average case.

As both best- and worst-case path lengths are in $O(\log N)$, the average-case length is in $\Theta(\log(N))$. Further simplification of the average cost will result in very loose bounds. Targeting practical settings, we calculate the average page lengths for various configurations and compare it to related work in Table 5.1. While this table is based on our theoretical results, the actual experimental results of r-ORAM height are presented in Figure 5.10.

Notice that our structure is a generalization of a binary tree for $x = 1$ and $y = 2$. Throughout this paper, the values x, y, and r equal the resulting optimal values given by Theorem 5.2.1.

5.2.2.4 r-ORAM Map addressing

In order to access a leaf in the r-ORAM structure, we have to create an encoding which uniquely maps to every leaf. This will enable us to retrieve the path from the root to the corresponding leaf

node. The encoding is similar to the existing ones in [65, 129, 135]. The main difference is the introduction of the new recursion, which we have to take into account. Every node in the outer or inner trees can have either two children in the same inner tree or/and two other children as a consequence of the recursion. Consequently, we need *two bits* to encode every possible choice for each node from the root of the outer tree to a leaf. For the non-recursive leaf trees, one bit is sufficient to encode each choice.

For tree-based ORAM constructions with full binary-trees, to map N addresses, a $\log N$ bit size encoding is sufficient for this purpose. This encoding defines the leaf tag to which the real element is associated.

In r-ORAM, we define a vector v composed of two parts, a variable-size part v_v and a constant-size part v_c, such that $v = (v_v, v_c)$. For the encoding, we will associate to every node in the outer and inner trees two bits. For every node in the leaf tree only one bit. Above, we have shown that the shortest path to a leaf node has length $r + \log(x)$ while the longest path has length $r \cdot \log(y) + \log(x)$. Consequently, for the variable-size vector v_v, we need to reserve at least $2 \cdot r$ bits and up to $2 \cdot r \cdot \log(y)$ bits for the worst case.

The total size of the mapping vector v, $|v| = |v_v| + |v_c|$, is bound by $2r + \log(x) \leq |v| \leq 2r \cdot \log(y) + \log(x)$, which is in $\Theta(\log(N))$. Figure 5.6 shows an address mapping example for two leaf nodes. The size of the block in the r-ORAM position map is upper bounded by $2 \cdot \log N$ bits. Finally, the mapping is stored in a position map structure following the recursive construction in [135]. To access the position map, the communication cost has, as in r-ORAM, a best-case cost of $O(\mathcal{B} \cdot \log^2(n) \cdot z)$ bits and worst-case cost of $O(\mathcal{C} \cdot \log^2(n) \cdot z)$ bits, where z is the number of entries. This complexity is in term of bits, not blocks. For larger blocks, we can neglect the position map. In Path ORAM or Shi et al. constructions, the size to access the position map is in $O(z \cdot \log^3 N)$ which is the result of accessing a path containing $\log N$ buckets a $\log N$ number of time. Each bucket has z blocks where each has size equal to $O(\log N)$.

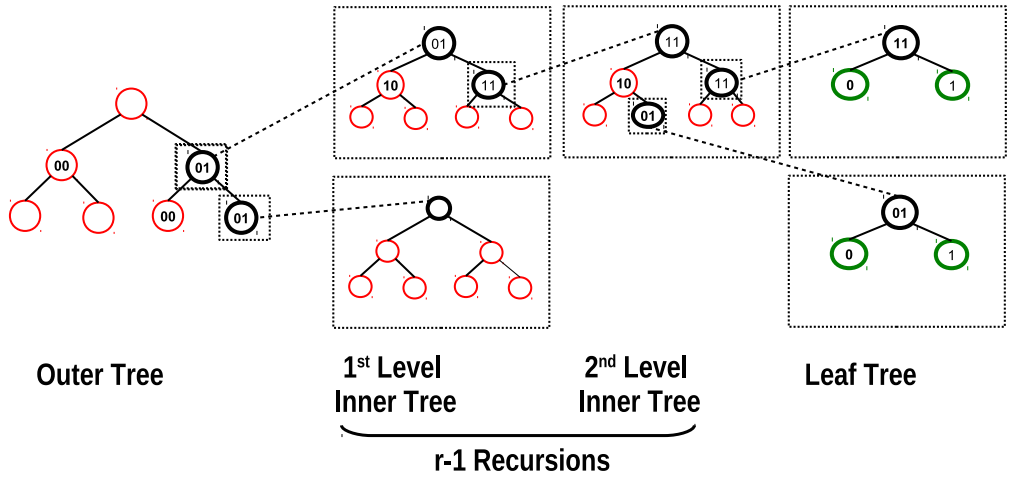

Figure 5.6: r-ORAM Map addressing

5.2.2.5 Communication complexity

First, we briefly formalize that the height can be seen as a multiplicative factor over all the recursion steps taking into consideration the eviction. Let N be the number of elements in the ORAM, denote by z the size of a bucket, β the position map factor, h the tree-structure height, l the block size and $\chi \geq 1$ the number of eviction, then for all tree-based ORAM the communication complexity C_T can be formulated as follows:

$$C_T = O(\underbrace{\chi \cdot z \cdot h \cdot l}_{\text{Data access}} + \underbrace{\beta \cdot z \cdot h \cdot \chi \cdot \log N}_{\text{Recursion}})$$

Reducing the height h decreases the entire communication overhead.

In this section, we are interested on computing the exact communication complexity (downloading/uploading) to access one block of size l. We will use for our computation the average height which is equal to $\approx 0.65 \cdot \log N$, see Table 5.1. In the following, we compute the communication complexities $C_{1,r}$ of r-ORAM over Path ORAM [135] and $C_{2,r}$ for r-ORAM over [129]. We denote the communication complexity for one access of Path ORAM and [129] by C_p and C_s. For an access, we download the entire path and upload it again. For Path ORAM, the eviction occurs at the same time when writing back the path. There is no additional overhead in the eviction. In the following equations, we take into consideration the variation of the bucket size. We later show in Section 5.2.4 that the size of r-ORAM applied to Path ORAM buckets increases by a factor of 1.2,

while it expectedly decreases by 30% if applied to [129]. The variation of the bucket size impacts the height reduction in both cases as follows:

$C_{1,r} \approx 2{\cdot}0.65{\cdot}\log N{\cdot}l{\cdot}z_{1,r}{+}\sum_{i=1}^{\lceil \frac{\log N}{\log \beta} \rceil -1} 2{\cdot}0.65{\cdot}z_{1,r}{\cdot}\log \frac{N}{\beta^i}{\cdot}\log \frac{N}{\beta^{i-1}} \approx 0.65{\cdot}\frac{z_{1,r}}{z_p}{\cdot}C_p = 0.78{\cdot}C_p.$

For [129]'s ORAM, for an eviction rate equal to 2, we are downloading 6 paths, plus the first one from which we have accessed the information. Thus, for each access, one has to download a total of 7 paths.

$C_{2,r} \approx 2 \cdot 0.65 \cdot 7 \cdot \log N \cdot l \cdot z_{2,r} + \sum_{i=1}^{\lceil \frac{\log N}{\log \beta} \rceil -1} 2 \cdot 0.65 \cdot 7 \cdot z_{2,r} \cdot \log \frac{N}{\beta^i} \cdot \log \frac{N}{\beta^{i-1}} \approx 0.65 \cdot \frac{z_{2,r}}{z_s} \cdot C_s \approx 0.5 \cdot C_s.$

In this result, we make use of an approximation due to the size of the position map. In Section 5.2.2.4, we have shown that that to map an element, approximately $2 \cdot \log N$ bits is needed instead of $\log N$. We will show that these results match the experimental results in Section 5.2.5.

5.2.3 κ-ary Trees

So far, we have used a *binary* tree for the recursion in r-ORAM, i.e., leaf and inner trees are full binary trees. In this section, we extend r-ORAM to κ-ary trees, cf. [65]. Generally, the usage of κ-ary trees reduces the height by a multiplicative factor equal to $\frac{1}{\log(\kappa)}$. For example, if we choose a branching factor $\kappa = \log N$, the communication complexity decreases by a multiplicative factor equal to $\log(\log N)$. We will now show that applying r-ORAM to a κ-ary tree will further decrease the communication complexity compared to the original κ-ary construction.

For parameters x and y defined above, the number of elements N can be computed by calculating the number of nodes in the outer and inner κ-ary tree for a recursion factor r:

$$\begin{aligned} N &= (\sum_{i=0}^{\log_\kappa y} \kappa^i - 1)^r \cdot x = (\frac{1-\kappa^{1+\log_\kappa y}}{1-\kappa} - 1)^r \cdot x \\ &= (\frac{\kappa}{\kappa - 1} \cdot (y-1))^r \cdot x \end{aligned} \tag{5.15}$$

Th. 5.2.3 shows how one should choose the recursion factor r, the height of the inner trees $\log y$ and leaf trees $\log x$ to minimize the cost of reading a path of κ-ary r-ORAM structure. In section 5.2.2.5, we have shown that the height factors over the total communication overhead reduction. Thus, any reduction applies for the the entire communication overhead computation.

Also, we show in Section 5.2.4 based on our security analysis that r-ORAM's bucket size over [65]'s ORAM decreases, thereby decreasing communication cost even more.

Theorem 5.2.3. *Let f be a decreasing function in κ with a range in $\mathbb{R}^+$. If $r = \log_\kappa((\frac{N}{\kappa})^{\frac{1}{f(\kappa)}})$, $x = 2$, and $y = \frac{\kappa-1}{\kappa} \cdot (\frac{N}{\kappa})^{\frac{1}{r}} + 1$, the optimum values for the best and worst-case cost equal*

$$\mathcal{C} = 1 + \log_\kappa((\frac{N}{\kappa})^{\frac{1}{f(\kappa)}}) \cdot \log_\kappa((\kappa-1) \cdot \kappa^{f(\kappa)-1} + 1), \text{ and}$$

$$\mathcal{B} = 1 + \frac{1}{f(\kappa)} \cdot \log_\kappa(\frac{N}{\kappa}).$$

The decreasing function f depends on the choice of κ, the branching factor. For $\kappa = 4$, $f(4) \approx 2$, while for $\kappa = 16$, $f(16) \approx 1.6$. The proof of the Theorem 5.2.3 is similar to the proof of Theorem 5.2.1, so we will only provide a sketch, highlighting the differences.

(Sketch). The first step in the proof is to represent the number of leaves x as a function of N, y, r, and κ the branching factor. That is, we reduce the number of variables in our optimization problem by one. Taking the logarithm of Eq. (5.15) leads to $\log_\kappa(x) = \log_\kappa(\frac{\kappa-1}{\kappa}N) - r \cdot \log_\kappa(y-1)$. Since our first goal is the minimization of the worst-case cost, we substitute $\log_\kappa(x)$ in the worst-case cost Eq. (5.22) by the value computed in the above equation and minimize the new expression. Note that the logarithm is base κ instead of 2 in the worst-case cost formula.

For simplicity, we consider the branching factor as a (given) constant, as it has an impact on the overflow probability. So, we assume a fixed branching factor matching a given bucket size. Finally, we follow the same steps as the proof of Theorem 5.2.1 to find the optimal recursive factor r, the number of leaf tree leaves x, and the number of inner/outer tree leaves y.

■

Example: For $\kappa = 4$, the optimal values for the best and worst-case cost respectively equal $\mathcal{B} \approx 0.55 \cdot \log_\kappa N$ and $\mathcal{C} \approx 0.95 \cdot \log_\kappa N$.

5.2.4 Security Analysis

5.2.4.1 Privacy Analysis

Theorem 5.2.4. *r-ORAM is a secure ORAM following Definition 2.3.4, if every node (bucket) is a secure ORAM.*

Proof (Sketch). If the ORAM buckets are secure ORAMs, we only need to show that two access patterns induced by two same-length sequences $\overrightarrow{a}$ and $\overrightarrow{b}$ are indistinguishable. To prove this, we borrow the idea from [135] and show that the sequence of tags t in an access pattern is indistinguishable from a sequence of random strings of the same lengTheorem

To store a set of N elements, r-ORAM will comprise N leaves and N different paths. During *Add* and *ReadAndRemove* ORAM operations, tags are chosen uniformly and independently from each other. Since the access pattern $A(\overrightarrow{a})$ induced by sequence $\overrightarrow{a}$ consists of the sequence of tags (leaves) "touched" during each access, an adversary observes only a sequence of strings of size $\log N$, chosen uniformly from random. The nodes in r-ORAM are bucket ORAMs, i.e., for an ORAM operations they are downloaded as a whole, IND-CPA re-encrypted, and uploaded exactly as in related work, they are secure ORAMs.

■

5.2.4.2 Overflow probability

To show that our optimization is a general technique for tree-based ORAMs, we compute the overflow probabilities of buckets and stash for both constant and poly-logarithmic client memory schemes. Specifically, we analyze r-ORAM for the constructions by [129], [65], and [135]. Surprisingly, for the first scheme, we are able to show in Theorem 5.2.5 that r-ORAM will reduce the bucket size while maintaining the exact same overflow probability. This is significant from a storage and communication perspective: it shows that r-ORAM can improve storage and communication overhead not only due to a reduction of the number of nodes (as shown in Section 5.2.2.1 and 5.2.2.5), but also by reducing the number of entries in every bucket.

For the second scheme which uses a "temporary" poly-log stash during eviction (needed to compute the least common ancestor), we show in Theorem 5.2.6 that r-ORAM offers improved communication complexities and a slightly better bucket size.

Finally for Path ORAM, we prove that the stash size increases only minimally and remains small. In Theorem 5.2.7, we show that this small increase is outweighed by smaller tree height.

We now determine the ORAM overflow probability for two cases, (1) r-ORAM applied to the constant client memory approach, and (2) to the poly-log client memory approach. For the first case, we consider an eviction similar to the one used by [129]. That is, for every level, we will evict χ buckets towards the leaves, where χ is called the eviction rate. For the second case, we consider a deterministic reverse-lexicographic eviction similar to [65] and [61]. In particular, for the poly-logarithmic setting, we investigate the application of r-ORAM over two different schemes. The first case consists of the application of r-ORAM over the scheme by [65]. For this, we study the overflow probability of the buckets and we show that the recursive structure offers better bucket size bounds. The second case represents the application of r-ORAM over Path ORAM. We determine the overflow probability of the memory, dubbed *stash*, where each bucket in r-ORAM has a constant number of entries z. Using deterministic reverse-lexicographic eviction greatly simplifies the proof while insuring the same bounds as the ones in randomized eviction [135].

To sum up, we are studying three different cases. (1) r-ORAM over [129] construction, (2) r-ORAM over [65] construction and (3) r-ORAM over Path ORAM [135]. For the first two, we have to quantify the bucket size while for the third one we have to quantify the stash size and the size of the bucket as well. For each setting, an asymptotic value of the number of entries z is provided. The main difference between the computation of the overflow probability in r-ORAM and related work is the irregularity of path lengths of our recursive trees. To better understand the differences, we start by presenting a different model of our construction in 2-dimensions.

Description: A 2-dimensional representation of r-ORAM consists of putting all the recursive inner trees as well as the leaf trees in the same dimension as the outer tree. Consequently, the outer tree, the recursive inner trees, as well as the leaf trees will together constitute only one single tree we call the *general tree*. The main difficulty of this representation is to determine to which level a given recursive inner tree is mapped to in the general tree.

The general tree, by definition, will have leaves in different levels. This can be understood as a direct consequence of the recursion, i.e., some leaves will be accessed with shorter paths compared to others. Moreover, the nodes of the recursive trees will be considered as interior nodes of the general tree with either 4 children or 2 children. Any interior node of an inner or outer tree is a

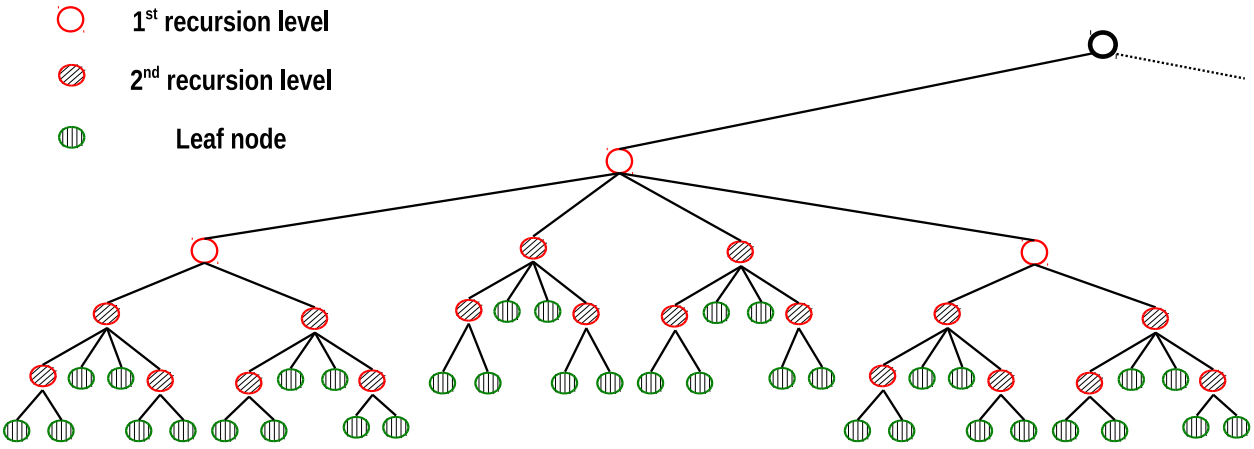

Figure 5.7: Structure of an r-ORAM

root for a recursive inner tree which means that any given interior node of an inner/outer tree has 2 children related to the recursion as well as another 2 children related to its inner/outer tree. These 4 children belong to the same level in our general tree.

Also, leaf nodes of inner or outer trees have only 2 children. Ultimately, we will have different distributions of interior nodes as well as leave nodes throughout the general tree. In the following, we will use the term of *interior node* as well as a *leaf node* in the proofs of our theorems to denote an interior or leaf node of the general tree. Figure 5.7 illustrates the topology of the general tree model of r-ORAM.

In the ith level, we may have leaf nodes as well as interior nodes. Also, the leaf/interior nodes reside in different levels with different non-uniform probabilities. Therefore, we will first approximate the distribution of the nodes in a given level of the r-ORAM structure by finding a relation between the leaf nodes and interior nodes of any level of r-ORAM. Then, we compute the relation between the number of nodes in the ith and $(i+1)$th level. This last step will help us to compute the expected value of number of nodes in any interior nodes in poly-log client memory scenarios. Finally we will conclude with the overflow theorems and their proofs for each scenario.

We present a relation between $I(i)$, the number of interior nodes, and $\mathcal{N}(i)$, the number of leaf nodes, for a level $i > r$, where r is the recursion factor. Notice that, for other levels $i \leq r$, there cannot be leaf nodes. Also, the leaves of the general tree are the leaves of the leaf trees. The maximum value of i equals the worst case $\mathcal{C}$.

Lemma 5.2.1. *Let* $f(r,x,y) = \frac{1+r\log(y)-r-2\log(x)}{2s^2}$ *and* $s > 0$. *For any* $i > r$, $e^{-f(r,x,y)} \leq \frac{I(i)}{\mathcal{N}(i)} \leq 2^{-\log(x)} \cdot r$.

Proof. First, we determine the number of interior nodes for $i > r$. In the same spirit as the proof of Theorem 5.2.2, we denote by A_j an array of $j \in [r]$ positions that has all positions initialized to zero. A_j represents the number of possible paths to a given level. The difference between counting the number of leaves and the number of interior nodes consists of the fact that an interior node may exist in any level without going through all recursions, i.e., it may happen that we reach a level without going through the last level of recursions. This means that elements of the array are equal to zero.

Counting of interior nodes boils down to divide Eq. (5.13) of Theorem 5.2.2 in r sub-equations, where each will count the number of ways to reach a specific level while all the positions of the array are still equal to 1. Therefore, the set of solutions of the following sub-equations has an empty set intersection.

$$\begin{aligned} A_1[1] - 1 &= i - 1 - \log(x), \\ (A_2[1] - 1) + (A_2[2] - 1) &= i - 2 - \log(x), \\ &\cdots \\ (A_r[1] - 1) + \cdots + (A_r[r] - 1) &= i - r - \log(x), \end{aligned}$$

where, for each $j \in [r]$, we have $1 \leq A_j[i] \leq \log(y)$.

Discussion: To have an intuition about these partitions, consider an example where $r = 4$ and $y = 16$. We have 4 sub-equations, where each represents the possible ways to reach an interior node in, e.g., the 4th level. The first array has only one position that can take values from 1 to 4. The first sub-equation will count the number of ways to get to an interior node at level 4 under the constraint that we have to stay in one recursion. In this case, the array can have only one value which is 4. For the second equation, we can have different combinations such as $(2,2)$, $(3,1)$, etc., but we do not have $(4,0)$, because it is already accounted for in the first sub-equation. We follow the same reasoning for the other sub-equations.

So, $I(i) = \mathcal{S}_1 + \cdots + \mathcal{S}_r$, the total number of solutions of the sub-equations. Also, we have $S_r \geq S_j$ for any $j \in [r-1]$, that is, $I(i) \leq r \cdot S_r$. From Theorem 5.2.2, we know that the number of solutions for the last equation S_r equals $2^{i-\log x} \cdot \mathcal{K}(i)$. Therefore, with the result of Theorem 5.2.2, we can conclude that $I(i) \leq 2^{i-\log(x)} \cdot r \cdot \mathcal{K}(i)$.

Also from Theorem 5.2.2, the number of leaves $\mathcal{N}(i) = 2^i \cdot \mathcal{K}(i)$. This leads to our first inequality $\frac{I(i)}{\mathcal{N}(i)} \leq 2^{-\log(x)} \cdot r$.

For our second inequality, notice that for any interior node of any level $i > r$, $I(i) \geq \frac{\mathcal{N}(i+1)}{2}$. This follows from the property that the ancestors of leaves in the $(i+1)$th level are interior nodes in the upper level. Using equality $\mathcal{N}(i) = 2^i \cdot \mathcal{K}(i)$,

$$\begin{aligned} \frac{I(i)}{\mathcal{N}(i)} &\geq \frac{\mathcal{N}(i+1)}{2\mathcal{N}(i)} \\ &= \frac{\mathcal{K}(i+1)}{\mathcal{K}(i)}. \end{aligned}$$

We have previously shown that $\mathcal{K}$ can be approximated by a normal distribution, cf. Eq. (5.14). Using this approximation, we obtain $\frac{I(i)}{\mathcal{N}(i)} \geq e^{-\frac{1+2i-2r-2\log(x)-c}{2s^2}}$.

Finally, since $c = r(\log(y) - 1)$, we have for $s > 0$ $\frac{I(i)}{\mathcal{N}(i)} \geq e^{-\frac{1+r\log(y)-r-2\log(x)}{2s^2}}$. This concludes our proof.

■

We will now show that, once we have a relation between leaves and interior nodes of the same level, finding the relation between any nodes of two different levels will be straightforward. We write the number of nodes as a sum of leaf nodes and interior nodes, such that $L(i) = \mathcal{N}(i) + I(i)$. Recall that for $i \leq r$, we have $\mathcal{N}(i) = 0$. We write $\mu = \frac{L(i+1)}{L(i)}$ (this will represent the expected value of the number of real elements in any interior nodes in Theorem 5.2.6). We present our result in the following lemma.

Lemma 5.2.2. *Let* $\mu = \frac{L(i+1)}{L(i)}$ *and* $X(i) = 1 - \frac{\mathcal{N}(i)}{L(i)}$. *For* $1 \leq i \leq \mathcal{C}$, μ *is bounded by* $2 \cdot X(i) \leq \mu \leq 4 \cdot X(i)$.

Proof. This result follows from two observations. First, the total number of interior nodes for the ith level is always larger than the total number of nodes in the $(i+1)$th level divided by 4. The

second observation is that the total number of interior nodes for the ith level is always smaller than the total number of nodes in $(i+1)$th divided by 2. Consequently, $\frac{L(i+1)}{4} \leq I(i) \leq \frac{L(i+1)}{2}$.

The second inequality follows from r-ORAM's structure where every interior node v has at least 2 children and at most 4 children. The recursion as previously represented in a 2-dimensional plane where an interior node in the outer or inner tree has 4 children, and every leaf node has exactly 2 children. So, every level has at least twice the number of interior nodes of the previous level.

We bound μ by algebraic transformations:

$$\frac{L(i+1)}{4} + \mathcal{N}(i) \leq L(i) \leq \frac{L(i+1)}{2} + \mathcal{N}(i)$$

$$\frac{\mu}{4} + \frac{\mathcal{N}(i)}{L(i)} \leq 1 \leq \frac{\mu}{2} + \frac{\mathcal{N}(i)}{L(i)}.$$

Finally, $2 \cdot (1 - \frac{\mathcal{N}(i)}{L(i)}) \leq \mu \leq 4 \cdot (1 - \frac{\mathcal{N}(i)}{L(i)})$.

■

From this result, for $i \leq r$, we have $2 \leq \mu \leq 4$, as $\mathcal{N}(i) = 0$.

We are now ready to present our three main theorems: the first one will tackle the constant client memory setting, and we compute the overflow probability of interior nodes. The overflow probability computation for leaf nodes, either for constant client memory or with poly-log client memory, is similar to the one presented by [129], based on a standard balls-into-bins argument. We omit details for this specific case. The last two theorems tackle tree Based ORAM constructions with memory.

Constant client memory: First, we compute the overflow probability of interior nodes. Then, a corollary underscoring the number of entries z will be presented.

Theorem 5.2.5. *For eviction rate χ, if the number of entries in an interior node is equal to z, the overflow probability of an interior node in the ith level is at most θ_i^z, where, for $i \leq r$ and $s = \lceil \log_4(\chi) \rceil$, $\theta_i = \frac{2^s}{2\chi}$, and for $i > r$: $\theta_i = \frac{2^s}{2\chi} \cdot (\frac{1}{1+\frac{x}{r}})^{i-r}$.*

Proof. The buckets of r-ORAM can be considered as queues [76]. Every bucket at the ith level has its service rate η_i and its arrival rate λ_i. The probability that the bucket contains z elements is given

by: $p(z) = (1 - \rho_i) \cdot \rho_i^z$, where $\rho_i = \frac{\lambda_i}{\eta_i}$. This is a result of M/M/1 queues [89]. The probability that the bucket will have strictly less than z elements equals $\sum_{i=0}^{k-1} p(i) = 1 - \rho_i^z$. The probability to overflow (to have more than z elements) equals ρ_i^z. In the following, it suffices to compute ρ_i for every level in our r-ORAM structure.

Consider eviction rates that are powers of 2. Then, for $i \leq \lceil \log_4(\chi) \rceil$, we have $\eta_i = 1$ and $\lambda_i \leq \frac{1}{2^i}$ (because for level 1 and deeper, buckets may have up to 4 children).

For $i > \lceil \log_4(\chi) \rceil$, the chance that a given bucket will be evicted is equal to $\eta_i = \frac{\chi}{I(i)}$, where $I(i)$ is the number of interior nodes in the ith level.

$\lambda_i = \frac{I(i)}{L(i+1)} \cdot \Pr(\text{parent gets selected}) \cdot \Pr(\text{parent is not empty})$, such that $\Pr(\text{parent gets selected}) = \eta_{i-1}$ and $\Pr(\text{parent is not empty}) = 1 - p_{i-1}(0) = \rho_{i-1}$. The ratio $\frac{I(i)}{L(i+1)}$ denotes the probability for a real element to be evicted, in the case of a binary tree the ratio is equal to $\frac{1}{2}$. Then, we have $\lambda_i = \frac{I(i)}{L(i+1)} \cdot \lambda_{i-1}$. By induction, the arrival rate equals $\lambda_i = \frac{1}{L(i+1} \cdot \frac{I(i) \cdot I(i-1) \cdots I(s+1)}{L(i) \cdot L(i-1) \cdots L(s+1)} \cdot I(s) \cdot \lambda_s$, where $s = \lceil \log_4(\chi) \rceil$. With $\lambda_s \leq \frac{1}{2^s}$ and $I(s) \leq 4^s$ (because we can have at most 4 children for every interior node), this equation can be upper-bounded such that:

$$\lambda_i \leq \frac{2^s}{L(i+1)} \cdot \frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} \cdots \frac{1}{1 + \frac{\mathcal{N}(s+1)}{I(s+1)}}. \tag{5.16}$$

We need to simplify the above inequality. First, notice that for every $s < i \leq r$

$$\frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} = 1, \tag{5.17}$$

because $\mathcal{N}(i) = 0$ (there is no leaf node for $i \leq r$). For $i > r$, using the result of Lemma 5.2.1.

$$\frac{1}{1 + \frac{\mathcal{N}(i)}{I(i)}} \leq \frac{1}{1 + \frac{x}{r}}, \tag{5.18}$$

where x is the number of leaves. For buckets at level $i > r$, we plug the result of equations 5.17 and 5.18 in 5.16 and we divide by the service rate η_i such that $\rho_i \leq \frac{I(i)}{L(i+1)} \cdot (\frac{1}{1+\frac{x}{r}})^{i-r} \cdot \frac{2^s}{\chi}$.

From Lemma 5.2.2, we have shown that $\frac{I(i)}{L(i+1)} < \frac{1}{2}$, because there are at least twice more nodes than interior nodes in the upper level (they may be leaves or interior nodes). Then $\rho_i \leq (\frac{1}{1+\frac{x}{r}})^{i-r} \cdot \frac{2^s}{2\chi}$. In this case ρ_i is upper-bounded by $\theta_i = (\frac{1}{1+\frac{x}{r}})^{i-r} \cdot \frac{2^s}{2\chi}$, and the overflow probability is then equal to θ_i^z.

For $i \leq r$, there are no leaves (i.e. $\mathcal{N}(i) = 0$), and the arrival rate is always bounded from Eq. 5.16 such that $\lambda_i \leq \frac{2^s}{L(i+1)}$.

Consequently, dividing by η_i and using the result of Lemma 5.2.2 $\frac{I(i)}{L(i+1)} < \frac{1}{2}$ we get $\rho_i \leq \frac{2^s}{2\chi}$. Considering $\theta_i = \frac{2^s}{2\chi}$ for $i \leq r$ concludes our proof.

■

In practice, the eviction rate χ equals 2. So, s is then equal to 1. In this case, the number of entries z in each bucket has the following size.

Corollary 5.2.1. *r-ORAM with N elements overflows with a probability at most $\omega \ll 1$ if the size of each interior bucket z in the ith level equals $\log \frac{N}{\omega}$ for $i \leq r$ and $z \approx \frac{1}{i-r+1} \cdot \log \frac{N}{\omega}$ for $i > r$.*

Sketch. By applying the union bound over the entire r-ORAM interior buckets, the probability of overflow is at most $N \cdot \theta_i^z$. Setting this value to the target overflow ω gives us the results for both underlined cases in Theorem 5.2.5. For the second equality, the approximation follows from the remark $\log(1 + \frac{x}{r}) < 1$, since $x \leq r$ in our optimal setting of Theorem 5.2.1.

■

The size of the internal buckets in r-ORAM are smaller compared to those of [129] by a multiplicative factor of approximately $\frac{1}{i-r+1}$ for $i > r$.

For $\omega = 2^{-64}$, $N = 2^{20}$, and $r = 7$, the size of the bucket equals 84 blocks for $i \leq 7$ while for, e.g., $i = 11$, the bucket size equals ≈ 17 blocks. For $i \leq r$, the bucket size is equal to the constant client memory construction, i.e., in $O(\log \frac{N}{\omega})$.

Poly-logarithmic client memory: Let us now tackle the case where r-ORAM is applied over tree ORAMs with poly-logarithmic client memory. For this, we consider two scenarios. The first deals with r-ORAM applied over [65]'s ORAM. The second one deals with r-ORAM over Path ORAM. In both cases, our overflow analysis is based on a deterministic reverse lexicographic eviction.

Theorem 5.2.6 determines the overflow probability of buckets in r-ORAM over [65] scheme. For each access, the eviction is done deterministically independently of the accessed paTheorem We show that the overflow probability varies for buckets in different levels due to the interior/leaf

node distribution. The parameter δ represents the unknown that should be determined for a given (negligible) overflow probability.

Theorem 5.2.6. *Let* $f(r,x,y) = \frac{1+r\log(y)-r-2\log(x)}{c}$*, and* $c > 0$*. For any* $\delta > 0$*, for any interior node* v*, the probability that a bucket has size at least equal to* $(1+\delta)\cdot\mu$ *is at most* $e^{-\frac{\delta^2\cdot\mu}{2+\delta}}$*, where* $F_1 \leq \mu \leq F_2$*.*

For $i \leq r$*:* $F_1 = 2$ *and* $F_2 = 4$*,*

for $i > r$*:*

$$F_1 = 4\cdot(1-\frac{1}{1+2^{-\log(x)}\cdot r}) \text{ and } F_2 = 2\cdot(1-\frac{1}{1+e^{-f(x,y,r)}}),$$

Proof. Let us fix an interior node v in r-ORAM belonging to the ith level. We are interested in the behavior of the node's load after a number of operations including eviction and adding operations. Let $L(i)$ denote the number of nodes residing in the ith level of the r-ORAM tree (these include the interior and the leaf nodes). Since the eviction is reverse-lexicographic and deterministic, we are sure that any element inserted before the time interval $\mathcal{T} = [t - L(i+1) + 1, \cdots, t]$ has been evicted from the ith level. Therefore, if we denote the number of elements residing in the node v, $S_t(v)$, we are sure that $S_t(v) = 0$ just a step before the interval $\mathcal{T}$. Consequently, it remains to determine the load of the interior node v for all the steps of the interval $\mathcal{T}$, i.e., the load of the node v in the (possible) presence of at most $L(i+1)$ elements in the ith level or above. Let us associate for every element j in $\mathcal{T}$ a random indicator variable χ_j which is equal to 1 if the element was assigned a path going through the interior node v. All elements in $\mathcal{T}$ are i.i.d. and their assignment probability is $\Pr(\chi_j = 1) = \frac{1}{L(i)}$. We have also $S_t(v) \leq \sum_{j\in[I(i+1)]}\chi_j$, which follows from the fact that all elements inserted in the interval $\mathcal{T}$ may *at most* all of them be assigned paths that go through v. In order to apply Chernoff's bound, we calculate the expected value of the sum of the indicator variables

$$E(\sum_{j\in[L(i+1)]}\chi_j) = \mu = \frac{L(i+1)}{L(i)}.$$

The exact value cannot be determined without computing the number of nodes existing in the ith level. What we can do is computing a tight bound of the expected value and then apply the Chernoff bound. Note that this expected value will be different from one level to the other.

Lemma 5.2.2 gives a bound on the expected value. This bound involves a relation between the leaf node and the interior nodes of the given level that we have computed in Lemma 5.2.1. For $i \leq r$, from Lemma 5.2.2, we know that $2 \leq \mu \leq 4$. For $i > r$, plug the first lemma in the second:

$$\underbrace{2 \cdot (1 - \frac{1}{1 + e^{-f(x,y,r)}})}_{F_1} \leq \mu \leq \underbrace{4 \cdot (1 - \frac{1}{1 + 2^{-\log(x)} \cdot r})}_{F_2}$$

Now, wrapping up with Chernoff's bound, for any $\delta > 0$ and for both cases $\Pr(S_t(v) \geq (1 + \delta) \cdot \mu) \leq \Pr(\sum_{j \in [L(i+1)]} \chi_j \geq (1 + \delta) \cdot \mu) \leq e^{-\frac{\delta^2 \cdot \mu}{2+\delta}}$. This concludes our proof.

■

To get an idea about the values of F_1 and F_2, we calculate them for $N = 2^{32}$: $F_1 = \frac{2}{5}$ and $F_2 = 3.42$. The theorem above represents a general bound to understand the overflow probability behavior. Since the expected value μ varies depending on the level, buckets sizes vary on every level. Consequently, fixing the expected value for every level results in much better bounds. For example, if for level i, $\mu = 1$, then the the probability of overflow with a bucket size equal to $64 = 1 + \delta$ is at most 2^{-88}, while for $\mu = 4$, the probability of overflow with the same bucket size is equal to 2^{-82}.

Corollary 5.2.2. *Le μ_i be the expected size of buckets in the ith level. r-ORAM with N elements overflows with a probability at most ω, if the size of each interior bucket z in the ith level equals $\mu_i + \ln \frac{N}{\omega}$ for $F_1 \leq \mu_i \leq F_2$.*

Proof (Sketch). By using the union bound, the probability that the system overflows equals $\omega = N \cdot e^{-\frac{\delta^2 \cdot \mu}{2+\delta}}$. This is a quadratic equation in δ that has one valid root (non-negative) approximately equal to $\frac{1}{\mu_i} \cdot \ln \frac{N}{w}$, where μ_i is the expected value of ith level. The size of the bucket in this case equals $z = (1 + \delta) \cdot \mu_i = \mu_i + \ln \frac{N}{\omega}$.

■

For r-ORAM over Path ORAM [135] with a deterministic reverse-lexicographic eviction [61], Theorem 5.2.7 calculates the probability of stash overflow for a fixed bucket size. The goal of this theorem is to determine the optimal bucket size and therefore the stash size for a fixed overflow probability.

Theorem 5.2.7. *For buckets of size $z = 6$ and tree height $L = \lceil \log N \rceil$, the stash overflow probability computes to*

$$\Pr(\mathsf{st}(\textit{r-ORAM}_L^6) > R) \leq 1.17 \cdot 0.88^R \cdot (1 - 0.54^N).$$

Proof. To prove this theorem, we borrow two lemmas from [135], namely their lemmas 1 and 2. We begin by giving a short overview over these two lemmas. For details and proofs, we refer to [135]. The first lemma underlines that the state of r-ORAM$_L^z$ is equal to the state of r-ORAM$_L^\infty$ after post-processing with a greedy algorithm G. r-ORAM$_L^\infty$ is r-ORAM$_L^z$ with an infinite number of entries in each block. For r-ORAM$_L^\infty$, we do not need a stash, since buckets can hold an infinite number of blocks. Algorithm G process r-ORAM$_L^\infty$ to have the same bucket construction as in regular r-ORAM$_L^z$ with deterministic reverse lexicographic eviction. Let $X(T)$ be the number of real blocks in some subtree T and $\eta(T)$ the number of nodes in subtree T. Now, Lemma 2 by [135] states that st(r-ORAM$_L^z$) $> R$, *iff* there exists a subtree T such that $X(T) > \eta(T) \cdot z + R$. Combining the two lemmas results in

$$\Pr(\mathsf{st}(\text{r-ORAM}_L^z) > R) = \Pr(\mathsf{st}(G(\text{r-ORAM}_L^\infty)) > R) \tag{5.19}$$

$$\begin{aligned} &\leq \sum_{T \in \text{r-ORAM}_L^\infty} \Pr(X(T) > \eta(T) \cdot z + R) \\ &< \sum_{i=1}^{N} 4^i \max_{\{T|\eta(T)=i\}} \Pr(X(T) > i \cdot z + R). \end{aligned}$$

The second inequality follows from the fact that the number of subtrees in a full binary tree of N elements is upper bounder by the Catalan number $C_i < 4^i$. The upper bound in Theorem 5.2.7 might be tighter if we consider that r-ORAM contains fewer subtrees than the ones in a full binary tree.

We now bound $\max_{\{T|\eta(T)=i\}} \Pr(X(T) > i \cdot z + R)$.

First, to find an upper bound for Eq. 5.19, we compute the expected value of $X(T)$ for subtree T of r-ORAM$_z^\infty$. Note that $E(X(T)) = \sum_{i=1}^{\eta(T)} E(|B_i|)$, where $|B_i|$ is the size of a bucket B_i in T. In r-ORAM$_L^\infty$, the expected value of buckets changes between levels, following a well-defined

distribution of interior nodes. For ease of exposition, we now assume that all buckets have the worst bucket load. To show this, we have to take into account two cases.

(1) If a bucket is a leaf bucket, the load is binomially distributed, such that $E(|B_i|) = N \cdot 2^L = 1$.

(2) For an interior bucket on level i, we have shown in Theorem 5.2.6 that $E(|B_i|) = \mu$ and $F_1 \leq \mu \leq F_2$ (F_2 is equal to its maximal value 4).

For both cases, we can bound the expected value of the bucket's load: $\max\{1, F_1\} \leq E(B_i) \leq 4$. That is, for any bucket in T, we obtain $\eta(T) \cdot \max\{1, F_1\} \leq E(X(T)) \leq 4 \cdot \eta(T)$.

Let $\Psi = E((X(T))$, $\eta(T) = n$, and $\xi = \frac{n \cdot z + R - \Psi}{\Psi}$. Applying Chernoff's bound to $X(T)$, we get

$$\Pr(X(T) > n \cdot z + R) = \Pr(X(T) > (1+\xi)\Psi) \leq e^{\frac{-\xi^2}{2+\xi} \cdot \Psi}.$$

With some algebraic computations, it is easy to see that

$$\frac{(n \cdot (z-4) + R)^2}{\Psi} \leq \xi^2 \cdot \Psi$$

and

$$(\frac{n \cdot (z-4+8) + R}{\Psi})^{-1} \leq (\frac{n \cdot (z-4) + R + 2\Psi}{\Psi})^{-1} \leq (2+\xi)^{-1}.$$

For $z > 5$, we have

$$\begin{aligned} \frac{1}{8}(n \cdot (z-4) + R) \leq & \frac{\xi^2 \cdot \Psi}{2+\xi} \\ e^{\frac{-\xi^2}{2+\xi} \cdot \Psi} \leq & 0.88^R \cdot e^{-n \cdot (z-4)}. \end{aligned} \tag{5.20}$$

Combining Eq. 5.20 with Eq. 5.19 results in

$$\begin{aligned} \Pr(\mathsf{st}(\text{r-ORAM}_{\mathsf{L}}^{\mathsf{6}}) > R) < & 0.88^R \cdot \sum_{i=1}^{N} e^{-i(6-4-\ln 4)} \\ \approx & 1.17 \cdot 0.88^R \cdot (1 - 0.54^N) \end{aligned}$$

■

Discussion: The probability is negligible in R (since 0.88 ¡1 and $1 - 0.54^N \xrightarrow{\infty} 1$). So, for a fixed overflow probability $\omega \ll 1$, we have to define the corresponding value of R by solving

the equation $\omega = 1.17 \cdot 0.88^R \cdot (1 - 0.54^N)$. An r-ORAM stash with N elements overflows with probability at most $\omega \ll 1$, if the size of each bucket is 6, and the stash has size $R = \frac{1}{\ln 0.88} \cdot \ln \frac{\omega}{1.17 \cdot (1-0.54^N)}$. For large values of N, $R \in \Omega(\ln(\omega^{-1}))$.

We have made a number of approximations in our proof that slightly bias the choice of the bucket size and round the upper bound. We could improve our upper bound by a more accurate approximation of the number of subtrees in r-ORAM. Also, we assume the worst expected value for each bucket on all levels which is 4. Theorem 5.2.7 is valid for any bucket size $z \geq 6$.

5.2.5 Performance Analysis

We now analyze the behavior of r-ORAM when applied to different tree-based ORAMs. As a start, we compute the communication complexity of r-ORAM access, based on the average height, and estimate the monetary cost of access with r-ORAM on Amazon EC2 cloud storage infrastructure. This first part is based on our r-ORAM theoretical results above. For *all previous* binary tree-based ORAMs, the communication complexity for a number of elements is always constant for fixed N. With previous ORAMs, you must always download an entire path. Following our theoretical estimates, we go on to present our r-ORAM implementation results and compare with Path ORAM [135]. We compare both the average height and the resulting communication improvements, and, finally, also evaluate the behavior of the stash.

5.2.5.1 Theoretical Results

Even if the worst-case complexity is in $O(\log N)$, the underlying constants gained with r-ORAM are significant. Table 5.1 compares between the height of a binary tree as with [61, 101, 129, 135] and the height of r-ORAM. Also, we compare r-ORAM on κ-ary trees, instead of binary ones, and we show that the recursive κ-ary tree r-ORAM gives better performances in terms of height access and communication cost.

Table 5.1 has been generated using parameters from Theorems 5.2.1 and 5.2.3. This table compares only the complexity of accessing an element in the tree, i.e., going from the root to the leaf. It does not take the communication overhead of accessing the position map into account which we will deal with later. Moreover, Table 5.1 computes only the number and not the size of

Table 5.1: Tree height comparison

		Number of elements			
		2^{10}	2^{20}	2^{40}	2^{60}
Binary ORAM trees [61, 101, 129, 135]		10	20	40	60
Binary r-ORAM tree	Best case	5	8	16	23
	Average case	6	14	26	40
	Worst case	8	16	31	47
4-ary ORAM tree [61, 65, 135]		5	10	20	30
4-ary r-ORAM tree	Best case	3	6	11	16
	Average case	5	8	16	24
	Worst case	5	10	19	28

Table 5.2: Tree-based ORAM gain

	Gain in %		
	Best-case	Average-case	Worst-case
Binary ORAM trees [61, 101, 129, 135]	60	35	22.5
4-ary ORAM trees [61, 65, 135]	45	20	5

nodes accessed. The overall communication complexities will vary from one scheme to the other, and we detail costs below, too. Table 5.2 shows the gain (in %) of r-ORAM applied to binary trees ORAM, not distinguishing whether a scheme has constant or poly-log memory complexity.

As shown in Table 5.2, we improve on average 35% when r-ORAM is applied to any binary tree ORAM and 20% when applied to 4-ary ORAM trees. Compared to binary trees, the gain for κ-ary trees is smaller due to the reduction of the height of the tree. Trees are already "flat", so the benefit of recursion diminishes.

We present the *total* communication overhead comparison and a monetary comparison of communication overhead between tree-based ORAM constructions (with constant and poly-log client memory). For this, we use blocks with size 1 KByte. The number of entries (blocks) in every node varies depeding on the scheme. We apply the result of Theorem 5.2.5 and Theorem 5.2.6 to vary

the size of the buckets accordingly. For the poly-logarythmic client memory, the size of the buckets of r-ORAM over Path ORAM are set to $z = 6$ based on Theorem 5.2.7. We take communication and storage overhead of the position map into account as well as the overhead induced by eviction (eviction rate equal to 2 for the constant client memory case).

Figure 5.8 depicts the communication cost per access, i.e., the number of bits transmitted between the client and the server for any read or write operation. The graph shows that r-ORAM applied to Path ORAM ($z = 6$) gives the smallest communication overhead. For example, with a dataset of 1 GByte, an access will cost 100 KByte in total. Moreover, if we set the number of entries z to 3 instead of 6, see [61], communication costs are divided by 2.

The storage overhead of tree-based ORAMs is still significant. Poly-log client memory ORAMs perform better, but still induce roughly a factor of 10. r-ORAM reduces this overhead down to a factor of 9.6, i.e., a reduction by 4%. For r-ORAM over [129] scheme, the saving is greater than 50% since we are reducing not only the height but also the size of the bucket.

Finally, we calculate the cost in US Dollar (USD) associated with every access, cf. Figure 5.9. As we obtain smallest communication overhead by using r-ORAM on top of Path ORAM, one would naïvely expect this to be the cheapest construction. However, Amazon S3 pricing is based not only on communication in terms of transferred bits (Up to 10 TB/month, 0.090 USD per GBytes), but also on the number of HTTP operations performed (GETs and PUTs), 0.005 USD per 1,000 requests for PUT and 0.004 USD per 10,000 requests per month for GET. Surprisingly, the construction by [65] with branching factor $\kappa = \log(N)$ is cheaper as it involves fewer HTTP operations compared to Path ORAM (however, in practice, the branching factor cannot be large since it will increase the size of the bucket).

5.2.5.2 Experimental Results

For a real-world comparison, we have implemented Path ORAM and r-ORAM including the position map in Python. Our source code is available for download [18]. Experiments were performed on a 64 bit laptop with 2.8 GHz CPU and 16 GByte RAM running Fedora Linux. For each graph, we have simulated 10^{15} random access operations. The standard deviation of the r-

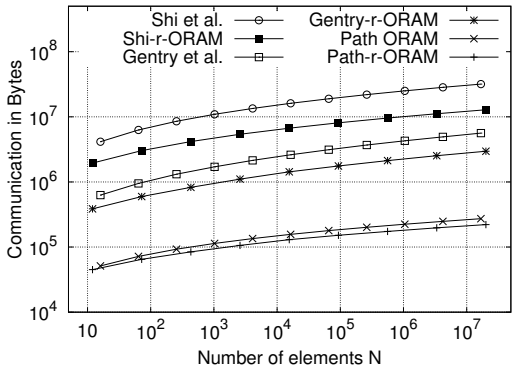

Figure 5.8: Communication per access

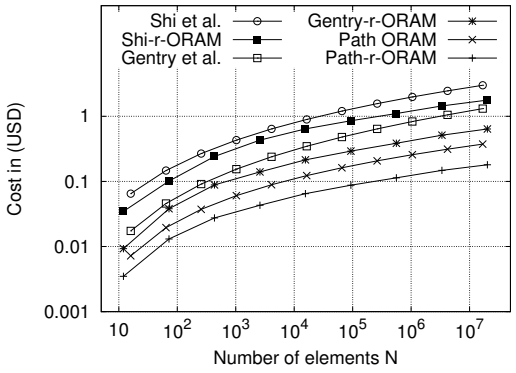

Figure 5.9: Communication cost per 100 accesses

ORAM height (communication complexity) was low at 0.015. The relative standard deviation for the average height (communication complexity) for 93312$\approx 2^{16.5}$ elements equals to 0.125.

The experiments begin with an empty ORAM. We randomly insert the corresponding number of elements. This step represents the initialization phase. Afterwards, we run multiple random accesses to analyze the height behavior and the stash size for r-ORAM over Path ORAM.

Figure 5.10 shows three curves: the height of binary tree ORAM (Path ORAM) from one hand and r-ORAM average and worst case height from the other hand. The height curves for r-ORAM are the result of 10^{15} accesses with a standard deviation of 0.015.

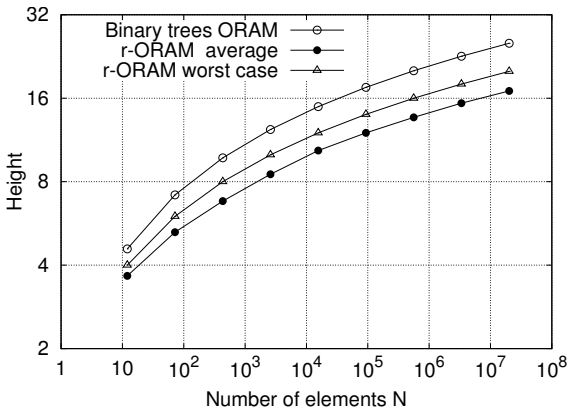

Figure 5.10: Average height comparison

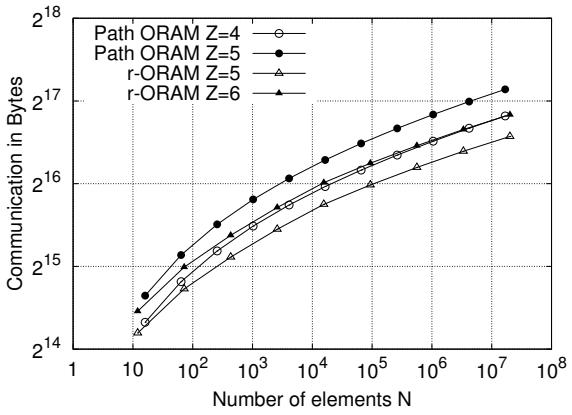

Figure 5.11: Communication per access

Our second comparison tackles communication including the recursion induced by the position map as well as the eviction per single access for different bucket sizes, see Figure 5.11. The eviction in r-ORAM is performed at the same time the path is written back. Also, we consider both

the upload/download phases. For example, with $N = 2^{14}$ and 4096 Bytes block size, the client has to download/upload 438 KByte with r-ORAM, instead of 640 KByte with Path ORAM, a ratio corresponding to the ratio of average heights, i.e., 31% of cost saving. Moreover, if we compare the curves associated to the minimum theoretical bounds for r-ORAM and Path ORAM, i.e., $z = 6$ and $z = 5$, the saving in terms of communication complexity is 20%. These curves represent the average of 10^{15} random accesses.

Finally, we measure r-ORAM's stash size for a number of random accesses between 2^{10} and 2^{20}. The number of operations represent a security parameter for our scenario, the more operations we perform the more likely the stash size increases. The upper bound of Th. 5.2.7 depends of the number of elements N, however for $N > 2$ the stash will have the same size independently of N because $1 - 0.54^N \approx 1$ for larger N. Thus, the stash in r-ORAM over Path ORAM has a logarithmic behavior in function of the security parameter, see Theorem 5.2.7.

Our experimental results confirm the upper bound given by Theorem 5.2.7, namely $R = \frac{1}{\ln 0.88} \cdot \ln \frac{\omega}{1.17 \cdot (1-0.54^N)} \approx \frac{1}{\ln 0.88} \cdot \ln \frac{\omega}{1.17}$. For example, for a probability of overflow equal to $\omega = 2^{-20}$,the security parameter here equals 20, the theoretical stash size R equals ∼110 blocks for any $N > 10$. In Figure 5.12, you can see that, for bucket size $z = 6$, we have exactly a logarithmic behavior as shown in the theorem. This figure shows the stash behavior based on the maximum, minimum, and median values. For a confidence level of 95%, the margin error is around 1.25. For 2^{20} operations, the maximum stash value equals 40 which is smaller than 110, the theoretical value, which is not surprising since some loose bounds have been used in the proof. The stash seems to increase logarithmically with the number of operations. However, theoretically the stash size behavior is not bounded. The graphs are logarithmic in the number of operations. In Figure 5.13, we show the average behavior of the stash size, to also indicate its logarithmic behavior.

Compared to Path ORAM with a similar bucket size, r-ORAM's stash requires up to 10 blocks more. This will not have any repercussion on the communication complexity. One might argue that the overall client memory size $\mathcal{M}$ has to be larger. However, the client memory size is defined as the stash *plus* the downloaded path during the operation such that $\mathcal{M} = R + \mathcal{P}$ where R is the stash size and $\mathcal{P}$ the number of blocks downloaded for a given path p. We have $\mathcal{P} = z \cdot |p|$

blocks where $|p| = \log N$ for Path ORAM and $|p| \approx 0.78 \cdot \log N$ for r-ORAM (worst-case). For a number of elements $N = 2^{20}$ and a bucket size $z = 5$, Path ORAM has to have 20 more blocks than r-ORAM and this will increase for greater number of elements.

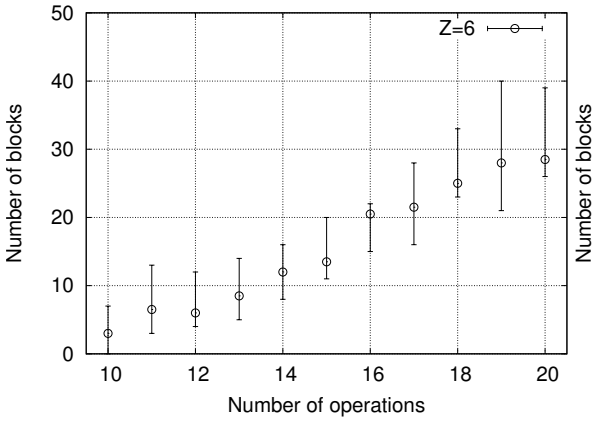

Figure 5.12: Stash size, $z = 6$, number of operations in $\log_2$

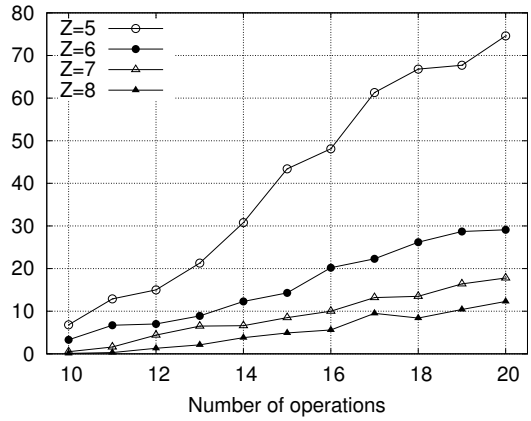

Figure 5.13: Average stash size, operations in $\log_2$

Section Conclusion. In this section, we have introduced a new recursive data structure that considerably improves upon previous works. The interesting aspect of this new data structure is in that it can be applied to future tree-based ORAM schemes. We show that r-ORAM can achieve 40% communication cost saving and 20% storage cost saving. One downside of r-ORAM is that it can be hard to implement. Moreover, an interesting problem to look at in future would be to add resizability to r-ORAM constructions as well. Finally, we are wondering whether this recursive data structure is the ideal one. Goldreich and Ostrovsky [70] showed that in the balls and bins model, at least a logarithmic blowup is required. None of the known constructions so far achieve this lower bound for any block size. It remains an open problem to find out the right data structure and memory shuffling procedure that can reach this lower bound with no additional assumptions about the client memory, block size or storage overhead.

5.3 Constant bandwidth ORAM

In this section, we present a new ORAM construction that achieves constant blow-up, i.e., to download a block, the client will download a constant number of blocks from the server. That is,

the cost of obliviousness becomes *nearly* for free contrary to previous constructions that at least introduce a poly-logarithmic blow-up. While previous constructions work in storage-only servers constant blowup ORAM constructions require the server to perform some additional computation on the data. Referring to ORAM schemes as constant ORAM constructions, however, is sometimes misleading, and one tends to think that this property is achievable with no assumptions which is not correct. To download a block using a constant ORAM scheme, the client exactly needs to spend $c \cdot B + \mathsf{poly}(N)$, where $c > 1$ is a constant and N is the number of blocks. An ORAM induces a constant blowup if (1) c is a constant independent of the choice of N and (2) the block size B needs to dominate the meta-information $\mathsf{poly}(N)$ such that $B \in \Omega(\mathsf{poly}(N))$. That is, if $B \in O(\mathsf{poly}(N))$, then it will perform similarly or even worse than known tree-based schemes. C-ORAM builds on the top of Onion ORAM [56] by introducing a new oblivious merge technique that reduces the communication complexity due to the meta-information but also the computation on the server side, which makes C-ORAM very appealing offering smaller block sizes.

5.3.1 Contribution Summary

Along with work on pure Oblivious RAM, Mayberry et al. [101] introduced the idea that communication overhead can be greatly reduced if the storage device is also considered to have some computational ability, which it generally does in a cloud setting. Using recent advances in homomorphic encryption, a small amount of computation on the server can be leveraged to cut a significant amount of communication to the client, see also [19].

Furthering this research, Devadas et al. [56] have recently proposed a hybrid ORAM-with-computation scheme that achieves $O(1)$ communication overhead. They achieve this by consecutively wrapping blocks in further layers of encryption as they proceed down the tree, effectively forming an "onion" out of the blocks. Unfortunately, it still has some major drawbacks:

1. Their scheme requires that the block has a very large size of $\Omega(\log^6 N)$. In practice, it can be up to 30 MB for reasonably sized databases.

2. The onion part of their scheme requires a large number of homomorphic multiplications, which are computationally very expensive. Depending on the encryption scheme used, overhead on the server may outweigh any communication saved.

In this work we tackle these problems. We start by showing that the homomorphic multiplications, and in fact the nesting "onion" nature of their solution, is not necessary. With careful application of an oblivious merging algorithm, all movement of blocks through the tree can be done with only homomorphic addition, resulting in a more computationally efficient algorithm. This also reduced the required block size by a $O(\log^2 N)$ factor and, as we will show, allows for $O(1)$ communication complexity in the worst case. Finally, we demonstrate via experimental evaluation that our scheme requires only a small storage overhead compared to Onion ORAM. For practical parameter values, we achieve significant improvement in block size and number of homomorphic operations. Table 5.3 summarizes our improvements when compared to Onion ORAM.

Table 5.3: Comparison of Onion ORAM and C-ORAM, containing block size, worst-case bandwidth, and number of homomorphic additions and multiplications. The simplified block value is a looser bound for easier comparison using $\lambda = \omega(\log N)$ and $\gamma = O(\lambda^3)$.

Scheme	Block size B	Simplified block size	Worst-case bandwidth	# multiplications
Onion ORAM	$\Omega(\gamma\lambda \log^2 N)$	$\Omega(\log^6 N)$	$O(1)$	$\Theta(B\lambda \log N)$
C-ORAM	$\Omega(\lambda[\log \lambda \log N + \gamma])$	$\Omega(\log^4 N)$	$O(1)$	$\Theta(B\lambda)$

5.3.2 Background: Onion ORAM

We start by briefly introducing the main idea of Onion ORAM [56] and then analyze its complexity to motivate our improvements.

5.3.2.1 Overview

An Oblivious RAM is a block-based storage protocol whereby a user can outsource some data to an untrusted server, and that server does not learn anything about the pattern of accesses that the user performs on that data. For instance, whether the user accesses the same block many times in a row, or each block individually in sequence, the server will not be able to distinguish between these two access patterns. In fact, a secure ORAM guarantees that *any* two access patterns will be

indistinguishable from the perspective of the server. This is accomplished by periodically moving, shuffling and reencrypting the data so that correlations between accesses are lost. A twist on that model introduced by [101], and used in Onion ORAM, is that instead of the traditional ORAM server definition where it only stores the data passively, Onion ORAM assumes that the server can also perform computations.

Onion ORAM is a tree-based ORAM, and shares many qualities with existing schemes [125, 129, 135]. Most importantly, data blocks are stored in a tree where each node of the tree is a "bucket" which contains some number of blocks. When blocks are added to the ORAM, they start at the root of the tree and are tagged as belonging to one of the leaf nodes. As the lifecycle of the ORAM continues, blocks percolate from the root to their assigned leaf node through a process called *eviction*. This way, a block can be located at any time by reading the path from its target leaf back to the root, since it is guaranteed to always reside on this path. The eviction process maintains a proper flow of blocks from the root to the leaves so that no buckets overflow with too many blocks. This is usually accomplished by picking a path in the tree, from root to a particular leaf node, and pushing all the blocks on that path as far as possible down the path toward the leaf node.

The contribution of Onion ORAM is then that it achieves constant communication complexity in the number of ORAM elements N, while only requiring polylogarithmic computation on the server. Although the client exchanges many pieces of data back and forth with the server, the key to having $O(1)$ communication complexity is that the size of one data block B dominates the communication. All other messages, ciphertexts etc. are collectively small compared to the actual data being retrieved. Therefore, it might be more intuitive to say that communication is $O(B)$. However, it is customary in ORAM literature to refer to the communication complexity in terms of multiplicative overhead, i.e., the cost compared to retrieving the same data without security. Everything is then divided by B, and we get to $O(1)$ communication complexity. Note that $O(1)$ communication complexity is not difficult if you allow unrestricted computation (FHE for instance achieves this trivially), so the limit to polylogarithmic computation is important.

The main idea behind Onion ORAM is an oblivious shuffling based on (computational) Private Information Retrieval (PIR). Therewith, ORAM read, write, and eviction operations can be performed without the client actually downloading data blocks and doing the merging themselves. This saves a huge amount of communication when compared to existing schemes like Path ORAM. Compared to existing tree-based ORAM schemes, Onion ORAM introduces a *triple eviction* that empties all buckets along the path instead of only pushing some elements down and leaving others at intermediate points in the tree. Elements in any evicted bucket will be pushed towards both children, thereby ensuring that after an eviction the entire evicted path is empty aside from the leaves. The authors take advantage of the fact that if you choose which path to evict by *reverse lexicographic ordering*, then you are always guaranteed during an eviction that the sibling of every node on your path will already be empty from a previous eviction. This allows for the entire process to be done efficiently and smoothly, because the entire contents of a parent can be copied into the empty bucket.

This triple eviction is accomplished by sending a logarithmic number of oblivious shuffling vectors to the server. These vectors, encrypted with an additively homomorphic encryption, obliviously map an old block of the parent bucket to a new position in the child. This operation is made by a matrix multiplication between the vector sent to the server and the bucket. Considering the size of the bucket as logarithmic, this algebraic computation should be performed a polylogarithmic number of times. This results that each block is encrypted, without transitional decryption, a logarithmic number of times, hence, the attributed name "onion".

The above results in an ORAM with constant communication complexity and constant client-memory in the number of elements N stored in the ORAM, see Table 5.3.

5.3.2.2 Analysis

As noted above, $O(1)$ communication complexity does not imply that blocks are the only information exchanged between client and server. In Onion ORAM, the client still needs to retrieve meta-information and send PIR vectors for PIR reads and PIR writes. Thus, Onion ORAM chooses the block size such that all communication between server and client is asymptotically dominated

by block size B. That is, if $B \in O(|\mathsf{meta} - \mathsf{information}| + |\mathsf{PIRvectors}|)$, then Onion ORAM has constant communication complexity.

Large Block Size: Consequently, to achieve constant communication complexity, Onion ORAM requires a large block size B. For a security parameter γ in the order of 2048 Bytes, bucket size $z = \Theta(\lambda)$, and number of elements N, the block size B in Onion ORAM is in $\Omega(\gamma\lambda \log^2 N)$. This is a significant increase over $B \in \Omega(\log N)$ as required by related work [129, 135]. Generally, large block sizes render ORAMs impractical for many real world scenarios where the block size is fixed and simply predetermined by an application. To mitigate the problem, Onion ORAM uses Lipmaa's PIR [95] instead of straightforward additively homomorphic PIR [91]. This decreases block size to $B \in \Omega(\gamma \log^2 \lambda \log^2 N)$. Factor λ is replaced by $\log^2 \lambda$. On a side note, observe that using Lipmaa's PIR might not result in much (or any) gain in practice. Parameter λ is a security parameter with $\lambda \in \omega(\log N)$. So, it is typically small and therefore "close" to $\log^2 \lambda$. For example, for $\lambda = 80$, $\log^2 \lambda = 40$ is in the same order of magnitude. Since Lipmaa's method requires substantially more computation than the straightforward approach, the small gain in communication is likely to be outweighed by additional computation time.

Onion ORAM block size example: For security parameter $\gamma = 2048$, number of elements $N = 2^{20}$, and security parameter $\lambda = 80$, the block size must be at least $B = 2048 \cdot \log^2(80) \cdot 20^2 \approx 33$ MBits. Thus, the dataset size equals $2^{20} \cdot 33 \cdot 10^6 \approx 35$ TBits. This computation is very rough and does not take into account additional, hidden constants such as the constant for the additively homomorphic cipher chunk in $\Omega(\gamma \log N)$, or smaller, yet still significant constants, like the fact that downloads have corresponding uploads which multiplies everything by 2. Requiring blocks of size at least 4 MBytes to store $N = 2^{20}$ elements is impractical for many real world applications. In conclusion, Onion ORAM can only be applied to very special data sets with very large block sizes.

5.3.3 Constant Communication ORAM

Overview: To achieve our increased efficiency and lower block size, we present a novel, efficient, oblivious bucket merging technique for Onion ORAM that replaces its expensive layered encryp-

tion. We apply our bucket merging during ORAM eviction. The content of a parent node/bucket and its child node/bucket can be merged obliviously, i.e., the server does not learn any information about the load of each bucket. The idea is that the client sends a permutation Π to the server. Using this permutation, the server aligns the individual encrypted blocks of the two buckets and merges them into a destination bucket. The client chooses the permutation such that blocks containing real data in one bucket are always aligned to empty blocks in the other bucket. As each block is encrypted with additively homomorphic encryption, merging two blocks is a simple addition of ciphertexts. For the server, merging is oblivious, because, informally, any permutation Π from the client is indistinguishable from a randomly chosen permutation.

For buckets of size $O(z)$, our oblivious merging evicts elements from a parent bucket to its child with $O(z \log z)$ bits of communication instead of $O(\gamma z^2)$ of Onion ORAM. As a result of applying our merging technique, we only need a *constant* number of PIR reads and writes for ORAM operations.

Based on our merging technique, we now present increasingly sophisticated modifications to Onion ORAM to reduce its costs. We call the resulting ORAM, i.e., Onion ORAM with our modification, C-ORAM. As a warm up, we present a technique allowing *amortized* constant communication complexity with a smaller block size B in $\Omega(z \log z \log N + \gamma z \log N)$. Our second and main technique achieves *constant* worst case communication complexity with smaller block size in $\Omega(z \log z \log N + \gamma z)$.

5.3.3.1 Oblivious Merging

Oblivious merging is a technique that obliviously lines up two buckets in a specific order and merges them into one bucket. Using this technique, we can evict real data elements from a bucket to another by permuting the order of blocks of one of them and then adding additively homomorphically encrypted blocks. Oblivious merging is based on an oblivious permutation generation that takes as input the *configurations* of two buckets and outputs a permutation Π. A configuration of a bucket specifies which of the blocks in the bucket are real blocks and which are empty. Permu-

tation Π arranges blocks in such a way that there are no real data elements at the same position in the two blocks.

5.3.3.2 C-ORAM Construction

C-ORAM keeps Onion ORAM's main construction. That is, C-ORAM is a tree-based ORAM composed of a main tree ORAM storing the actual data and a recursive ORAM storing the position map. The position map consists of a number of ORAM trees with linearly increasing height mapping a given address to a tag. For n elements stored in the ORAM, the communication needed to access the position map is in $O(\log^2 N)$. As with all recent tree-based ORAMS, the recursive position map's communication complexity is dominated by the block size. For the remainder of this paper, we therefore restrict our description to C-ORAM's main data tree.

Let N be a power of 2. C-ORAM is a binary tree with L levels and 2^L leave nodes. Each node/bucket contains $\mu \cdot z$ blocks. Here, z is the number of slots needed to hold blocks as in Onion ORAM and μ is a multiplicative constant that gives extra room in the buckets for noisy blocks, a detail we will cover below which is important for our construction. We maintain the same relation between N, L and z as in Onion ORAM, namely $N \leq z \cdot 2^{L-1}$. Each block in a C-ORAM bucket is encrypted using an additively homomorphic encryption, e.g., Pailler's or Damgard-Jurik's cryptosystem. Also, each bucket contains IND-CPA encrypted meta-information, *headers*, containing additional information about a bucket's contents.

Headers Bucket headers are an important component in C-ORAM as they determine how oblivious permutations are generated. A bucket header is comprised of two parts: the first part stores for each block whether it is noisy, contains real data or is empty. The second part stores the block tags. More formally, the header is composed of two vectors header_1 and header_2. Vector header_1 has length $\mu \cdot z$, and each element is either $\mathsf{noisy}, \mathsf{empty}$ or real. Thus, each element has a size of two bits. The total size of this vector is in $O(\mu z)$. header_2 is a $(\mu \cdot z \times \log N)$ binary matrix. The rows represent the address of the blocks. Finally, as with all tree based ORAMS, each block in a bucket also contains the encryption of its address. That is, the address of each block is encrypted separately from the block itself. We show a high level view of a C-ORAM bucket in Figure 5.14.

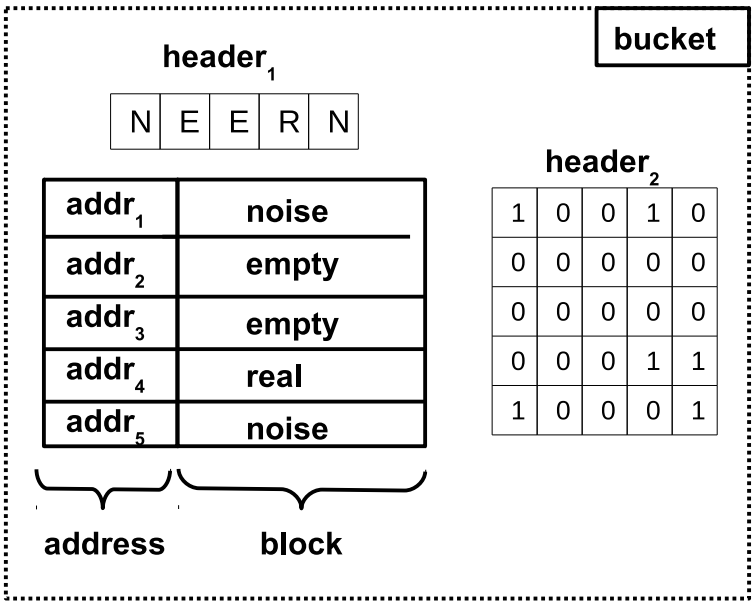

Figure 5.14: C-ORAM bucket structure

5.3.3.3 C-ORAM: First Construction

To prepare for our main contribution, we start by presenting a new technique allowing amortized constant communication complexity with a smaller block size.

Overview To access an element in C-ORAM, i.e., read or write, the client first fetches the corresponding tag from the position map. This tag defines a unique path starting from the root of the ORAM tree and going to a specific leaf given by the tag. The element might reside in any bucket on this path. To find this element, we make use of a PIR read [91] that will be applied to each bucket. To verify whether the block exists in a bucket, the client downloads the encrypted headers of each bucket. Therewith, the client can generate a PIR read vector retrieving the block from a bucket. To preserve the scheme's obliviousness, the client sends PIR read vectors for each bucket on the path. Once the block has been retrieved, the client can modify the block's content if required, then insert it back into the root of the C-ORAM tree using PIR write. This is the standard Path-PIR behavior to read from or write into blocks [101].

Eviction in our first construction takes place after every $\chi = O(z)$ access operations. As in Onion ORAM, a path in C-ORAM is selected following deterministic reverse lexicographic

order. Then, the entire root of the ORAM tree is downloaded, randomly shuffled and written back (additively homomorphically) encrypted. Finally, the eviction is performed by repeatedly applying an oblivious merge on buckets along the selected path. Any bucket belonging to this path is obliviously merged with its parent while the other child of the parent will be overwritten by a copy of the parent bucket. We call the former bucket on the path the *destination* bucket and the latter one its *sibling* bucket.

Before starting the eviction of a specific path, an invariant of the eviction process is that siblings of buckets of this path are empty, except the leaves. After the eviction, all buckets belonging to the evicted path will be empty except the leaf [56]. Note that siblings of this path, after the eviction, will not be empty anymore. See Figure 5.15 for a sample eviction with $N = 8$.

Sibling buckets, since they are simply copies of their parents, will contain blocks with tags outside the subtree of this bucket. These blocks are called *noisy* blocks as they do not belong into this subtree and are essentially leftover "junk". Now for correctness, in our construction, we will guarantee that the number of noisy blocks in any bucket is upper bounded. So, there will always be space for real elements in a bucket and will not overflow.

Elements in each bucket are encrypted using additively homomorphic encryption, respectively. Given two buckets B_1 and B_2, oblivious merging will permute the position of blocks in B_1 such that there are no real or noisy element at the same positions in B_1 and B_2. Consequently, if there is a real element in the ith position in B_1, then for the scheme to be correct, the ith position in B_2 should be empty. The following addition of elements at the same position in B_1 and B_2 will preserve the value of the real element. After χ operations, we also download the leaf bucket to delete its noisy blocks.

Details and Analysis Let $\mathcal{P}(tag)$ denote the path starting from the root and going to the leaf identified by tag. The path is composed of $L + 1$ buckets including the root. $\mathcal{P}(tag, i)$ refers to the bucket at the ith level of $\mathcal{P}(tag)$. For example, $\mathcal{P}(tag, 0)$ is the root bucket. $\mathcal{P}_s(tag, i)$ is the sibling of bucket $\mathcal{P}(tag, i)$. Let χ the period of eviction which is in $O(z)$. $\mathsf{Identity}$ is an empty

bucket containing only encryptions of zero. We also consider an additive homomorphic encryption scheme $\mathsf{SKE}^h = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$

Algorithm 3 presents details of the access operation. An access can be either an ORAM Read or a Write operation. The only difference between the two is that a write changes the value of the block before putting it back in the root. The access operation invokes a PIR read algorithm, see Algorithm 4 that obliviously retrieves a block. Algorithm 5 shows the eviction where elements percolate towards their leaves using oblivious permutations, see Algorithm 6.

Block size The following asymptotic analysis will be in function of z, N, and γ. z is the size of the bucket, N the number of elements, and γ the length of the ciphertext of the additively homomorphic encryption. The communication complexity induced by an ORAM access operation comprises a PIR read operation and the eviction process (happening every $\chi \in O(z)$ accesses). The size of the bucket is $\mu \cdot z$, but we will show in our security analysis section later that μ is a constant. Therefore, we ignore it in our analysis.

First, the client performs PIR reads $L + 1$ times. For this, the client has to download all addresses in the path, i.e., $O(z \cdot L \cdot \log N)$ bits. Also, the client should send a logarithmic number of PIR read vectors $\mathcal{V}$ with size $O(\gamma \cdot z \cdot L)$ bits. Note that the computation of PIR read vectors outputs, for all but one buckets' block, encryption of zeros. Instead of sending back a logarithmic number of blocks to the client, the server only sends a single block, the summation of all the blocks output, cf. Algorithm 3. Thus, the client only retrieves a single block B. A PIR read applied to all buckets of the path induces an overhead in $O(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + B)$.

For the eviction, the client downloads header_1 and the ith column of header_2 and sends permutations for all buckets in the path. Thus, the overhead induced by the permutations is $O(L \cdot z \cdot \log z)$ bits. Also, after every $\chi = O(z)$ operations, the client downloads the root and one leaf, which has $O(zB)$ communication complexity. Amortized, for each operation we have $O_z(B)$ communication complexity (amortized over z).

Algorithm 3: Access(op, adr, data, ctr, st): C-ORAM access operation, 1^{st} construction

Input: Operation op, address adr, data data, counter ctr, state st
Output: Block B associated to address addr
// Fetch tag value from position map
tag $=$ posMap(adr);
posMap(adr) $\xleftarrow{\$}$ $[N]$;
if ctr $= 0 \mod (\chi)$ **then**
  Download root bucket, refresh encryptions, randomize order of real elements;
  Evict(st);
else
  for i **from** 0 **to** L **do** $B = B+$PIR-Read(adr, $\mathcal{P}$(tag, i)) ;
end
if op $=$ write **then** set $B =$ data ;
ctr $=$ ctr $+ 1$;
Upload the encrypted block to root $\mathcal{P}(\text{tag}, 0)$;

In conclusion, each access has $O_z(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + z \cdot \log(z) \cdot L + B)$ communication complexity. To have constant communication complexity in B, the block size should be $B \in \Omega(z \cdot L \cdot \log N + \gamma \cdot z \cdot L + L \cdot z \cdot \log z) \in \Omega(\lambda \cdot \log^2 N + \gamma \cdot \lambda \cdot \log N)$.

The above is a consequence of $z = \Theta(\lambda)$, $\lambda \in \omega(\log n)$, and $L \in \Theta(\log N)$. Based on current attacks [95], $\gamma = O(\lambda^3)$. Therefore, $\lambda \cdot \log^2 N$ is dominated by $\gamma \cdot \lambda \cdot \log N$, and $B \in \Omega(\gamma \cdot \lambda \cdot \log N)$.

The block size of our first modification is already a $\log N$ multiplicative factor improvement over the block size of Onion ORAM. However, in practice, this value is still large. The main idea of our second construction is based on the following observation. The block size has exactly the same asymptotic as transmitted vectors $\mathcal{V}$. So to improve the block size, we change the way we are accessing the ORAM. Note that we can de-amortize C-ORAM first construction using techniques from [140].

5.3.3.4 C-ORAM: Second Construction

We start by further reducing the block size – again by a multiplicative factor of $\log N$ compared to our first construction. Recall that in our first construction, the worst case involves a blow-up of $O(z)$, because during eviction the client needs to download $O(z \cdot B)$ bits. In our second and main construction, the eviction remains exactly the same, and our focus will only be on ORAM access.

Algorithm 4: PIR-Read$(\mathsf{adr}, \mathcal{P}(\mathsf{tag}, \mathsf{level}))$

Input: Bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level}))$, address adr
Output: Block B
Retrieve and decrypt addresses Addr of bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level}))$;
// Compute the PIR-Read vector $\mathcal{V}$ in client side
if adr $\in$ Addr **then**
  // Retrieve the index α
  $\alpha = \mathsf{Addr}[\mathsf{addr}]$;
  for i **from** 1 **to** $\mu \cdot z$ **do**
    if $i \neq \alpha$ **then** $\mathcal{V}_i \leftarrow \mathsf{SKE}^h.\mathsf{Enc}(0)$ **else**
    $\mathcal{V}_i \leftarrow \mathsf{SKE}^h.\mathsf{Enc}(1)$;
  end
else
  for i **from** 1 **to** $\mu \cdot z$ **do** $\mathcal{V}_i \leftarrow \mathsf{SKE}^h.\mathsf{Enc}(0)$;
end
// Retrieve block in server side
Parse bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$ as $(\mu \cdot z \times |B|)$ binary matrix $\mathcal{M}$;
$B = (\sum_{i=1}^{\mu \cdot z} \mathcal{V}_i \cdot \mathcal{M}_{1,i}, \cdots, \sum_{i=1}^{\mu \cdot z} \mathcal{V}_i \cdot \mathcal{M}_{|B|,i})$;
Update $\mathsf{header}_1^{\mathsf{level}}$ of bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$;

Overview In our first modification, we perform a PIR read per bucket during an access. Contrary, we now perform an oblivious merge to find out the block to retrieve. For an ORAM access to tag, our idea is to perform a special evict of path $\mathcal{P}(tag)$. We push all *real* elements in $\mathcal{P}(tag)$ towards the leaf and then simply access the leaf bucket. So, we preserve access obliviousness and make sure that the element we want is pushed into leaf bucket tag.

This approach comes with several challenges. We must preserve the bucket distribution. That is, we have to maintain the empty sibling property, as guaranteed by the reverse lexicographic eviction, before evicting any path. Instead of deterministically selecting a path for eviction, we select randomly. However, with randomized eviction, we still have to guarantee empty siblings on the evicted path. By randomly evicting a path, we might copy a bucket in its sibling resulting in a correctness flaw.

Our approach will be to temporarily clone the path $\mathcal{P}(tag)$. The clone of $\mathcal{P}(tag)$ serves to simulate the eviction towards the leaf bucket, and we remove the clone after the access operation. We apply the oblivious merging on the bucket of this cloned path, and at the end we will have all

Algorithm 5: Evict(st), eviction process

Input: State st
Output: Evicted path and updated state st
for i **from** 0 **to** $L-1$ **do**
  Retrieve header_1^i and header_1^{i+1};
  Retrieve C_i and C_{i+1} respectively the ith and the $(i+1)$th column of header_2^i and header_2^{i+1} of the bucket $\mathcal{P}(\mathsf{st}, i)$ and $\mathcal{P}(\mathsf{st}, i+1)$;
  $\pi \leftarrow \mathsf{GenPerm}((\mathsf{header}_1^i, \mathsf{C}_i), (\mathsf{header}_1^{i+1}, \mathsf{C}_{i+1}))$, generate the oblivious permutation π;
  `// Merge the parent and destination bucket`
  $\mathcal{P}(\mathsf{st}, i+1) = \pi(\mathcal{P}(\mathsf{st}, i)) + \mathcal{P}(\mathsf{st}, i+1)$;
  if $i < L-1$ **then**
    `// Copy the parent bucket into its sibling`
    $\mathcal{P}_s(\mathsf{st}, i) = \mathcal{P}(\mathsf{st}, i)$;
  else
    `// Merge the last bucket with the sibling leaf`
    Retrieve header_1^{i+1} and C_{i+1} from the sibling leaf;
    $\pi \leftarrow \mathsf{GenPerm}((\mathsf{header}_1^i, \mathsf{C}_i), (\mathsf{header}_1{}^{i+1}, \mathsf{C}_{i+1}))$;
    $\mathcal{P}(\mathsf{st}, i+1) = \pi(\mathcal{P}(\mathsf{st}, i)) + \mathcal{P}(\mathsf{st}, i+1)$;
  end
  $\mathsf{Update}(\mathsf{header}_1^i)$ and store it with bucket $\mathcal{P}_s(\mathsf{st}, i)$;
  $\mathsf{Update}(\mathsf{header}_1^{i+1})$ and store it with bucket $\mathcal{P}(\mathsf{st}, i+1)$;
  $\mathcal{P}(\mathsf{st}, i) = \mathsf{Identity}$;
end

real elements in the leaf bucket of the cloned path. Finally, we apply a PIR read to retrieve the block.

Besides, to get rid of the amortized cost and have a scheme that only requires a constant bandwidth in the worst case, we make use of a PIR write operation that will be performed during every access. In the first construction, we have to shuffle the root bucket since oblivious merging has to be performed on random buckets for security purposes. Moreover, we need to eliminate noisy blocks from the leaf buckets and therefore after each χ operations, the client downloads the evicted leaf to eliminate all noisy blocks. In our second C-ORAM construction, we are evicting after every access. Consequently, we can be certain that the root bucket is always empty after an eviction. The first PIR write operation that we perform will randomly insert the block in an empty root bucket after any access obliviously. The second use of PIR write is to delete the retrieved element from the leaf. In fact, we can also delete noisy blocks by the same tool but a PIR read is needed to

retrieve first the noisy block that we will overwrite with a PIR write. We dedicate Section 5.3.4.2 to analyze security and correctness of our modification.

Details and Analysis Algorithm 7 presents the core of our second C-ORAM construction. Now, instead of performing a logarithmic number of PIR reads, we only invoke an Evict-Clone to read a block, cf. Algorithm 8. Evict-Clone uses our oblivious merging together with one PIR read to retrieve a block. We evict after every access. To eliminate noisy blocks that have been percolated to the leaf bucket, we use a PIR write to delete the noisy block, cf. Algorithm 9.

Block size The access operation in C-ORAM is composed of scheduled path eviction, eviction in the cloned path, a PIR read, and two PIR writes. The size of the headers are negligible compared to the PIR read and write vectors. For sake of clarity, we therefore avoid including them in our asymptotic analysis.

First, the eviction always involves an overhead of $O(zL \log z)$. Evict-Clone performs one PIR read in addition to the regular evict. Finally, we retrieve the block of size B. Therefore, the overhead induced by these steps is $O(zL \log z + z \log N + \gamma z + B)$. Adding the two PIR writes and single PIR read operation will not change asymptotic behavior since the number of these operations is constant in N. In conclusion, to have a bandwidth that is constant in block size B, the block size should be $B \in \Omega(z \cdot L \cdot \log z + z \cdot \gamma)$.

With $z \in \Theta(\lambda)$, $\lambda \in \omega(\log N)$ and $L \in \Theta(\log N)$, we achieve $B \in \Omega(\lambda \cdot [\log N \cdot \log \lambda + \gamma])$. In practice, $\gamma \in O(\lambda^3)$, so γ dominates $\log N \cdot \log \lambda$. Therefore, block size B is $B \in \Omega(\gamma\lambda)$.

Our second C-ORAM construction achieves worst-case constant blow-up and omits inefficient PIR reads performed for ORAM access. This second construction improves the block size by a multiplicative factor of $\log^2 N$ compared to Onion ORAM.

As you can see, the main overhead of C-ORAM's block size comes from the size of ciphertext γ. Recall that $\gamma \in O(\lambda^3)$. Therefore, the smaller the additively homomorphic ciphertext will get, the smaller the block size of C-ORAM will be.

5.3.4 C-ORAM analysis

5.3.4.1 C-ORAM correctness analysis

The goal of the correctness analysis section is to show that, during any eviction (algorithms Evict and Evict-Clone), the probability that a failure occurs is small. The failure in C-ORAM is defined as the lack of encryption of zeros in the evicted path. In this section, we only consider the proof of correctness of C-ORAM's first construction. The proof of correctness of C-ORAM's second construction is a straightforward extension from the first one. Before presenting details of our correctness analysis, we introduce some notations and assumptions.

Let $B_{i,j}$ refer to the bucket at the $i$thlevel of the path evicted at the jth step. Each bucket contains $\mu \cdot z$ blocks, with integer $\mu > 1$. In C-ORAM's first construction, the root bucket contains z real elements and $(\mu - 1) \cdot z$ empty blocks. We set $\phi = \mu - 1$. An empty block represents an additively homomorphic encryption of zero. Each bucket cannot have more than z real elements at any time with high probability, as we will prove in Theorem 5.3.3. Let $Z_{i,j}$ be the discrete random variable of the number of blocks containing an encryption of zero in bucket $B_{i,j}$. Similarly, $R_{i,j}$ represents the number of real blocks. Recall that if a real block is pushed to a path leading to a leaf different from its own tag, this block is called a noisy block. $\tilde{N}_{i,j}$ represents the random variable that counts the number of noisy blocks in bucket $B_{i,j}$. Finally, the jth eviction step is the eviction of the jth path following a deterministic reverse lexicographic order.

Formally, the eviction in Evict algorithm fails if $\exists i \in \{0, \ldots, L\}$ and $k \in \mathbb{N}$ such that $Z_{i+1,k} < R_{i,k}$ or $Z_{i,k} < R_{i+1,k}$. Thus, the proof's goal will be to show that there is no such integer $i \in \{0, \ldots, L\}$ that verifies both inequalities with high probability.

First, we need to introduce two properties that will help us to understand the proof and the eviction mechanism more thoroughly. The first property is called the *path composition history* while the second one is the *bucket composition history*. Given a path $\mathcal{P}(j)$, the *path composition history* captures the eviction step in which each bucket has been created. Given a bucket $B_{i,j}$, the *bucket composition history* is a sequence that captures all buckets that have contributed to the construction of the bucket $B_{i,j}$.

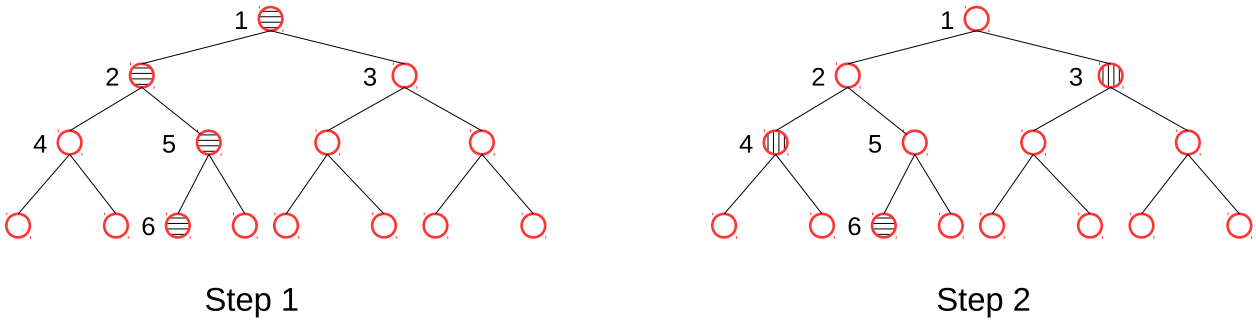

Figure 5.15: Buckets on evicted path are with horizontal hatching. Bucket 3 is a copy of the root. Bucket 4 results from merging buckets 1 and 2. Bucket 6 results from merging 1, 2, and 5.

Path composition history In C-ORAM, the eviction follows a deterministic reverse lexicographic order. In the $j$thstep of eviction, every bucket of the path $\mathcal{P}(j)$ has been created on a previous eviction. Thus, we associate to a bucket its *eviction step* during which it has been created. In particular, every bucket in this path has been created from a different eviction step. We are interested on defining the relation between the eviction steps of buckets belonging to the same evicted path. This relation follows a pattern which is common to all evicted paths. For instance, in Figure 5.16, the path $\mathcal{P}(9)$ of the 9^{th} eviction is composed of buckets $B_{1,8}, B_{2,7}, B_{3,5}$. These are buckets that were created, respectively, in the 8^{th}, 7^{th}, and 5^{th} eviction step. We do not count the root bucket and the leaf, because the pattern of their eviction is clear. That is, the root is evicted every time while the leaf is evicted following reverse lexicographic order.

Formally, for N elements stored in the ORAM and $L \in \Theta(\log N)$, one can easily show by induction that the $j$thevicted path, for all $j \geq 1$, is composed of $\{B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{L-1,j-2^{L-2}}\}$.

After L evictions, buckets belonging to an evicted path, except the leaves, are copies of a bucket from previous evictions. In our proof, we will later assume that the ORAM has performed a number of evictions larger than L. We will also consider the worst case where all buckets might eventually contain real or noisy blocks.

Bucket composition history This property follows from the previous one. Given a path $\mathcal{P}(j)$, the eviction will empty all buckets in this path except the leaf. The eviction works as follows: the root $B_{0,j}$ will be merged with its destination child $B_{1,j-2^0}$ in the path while the sibling $B'_{1,j}$,

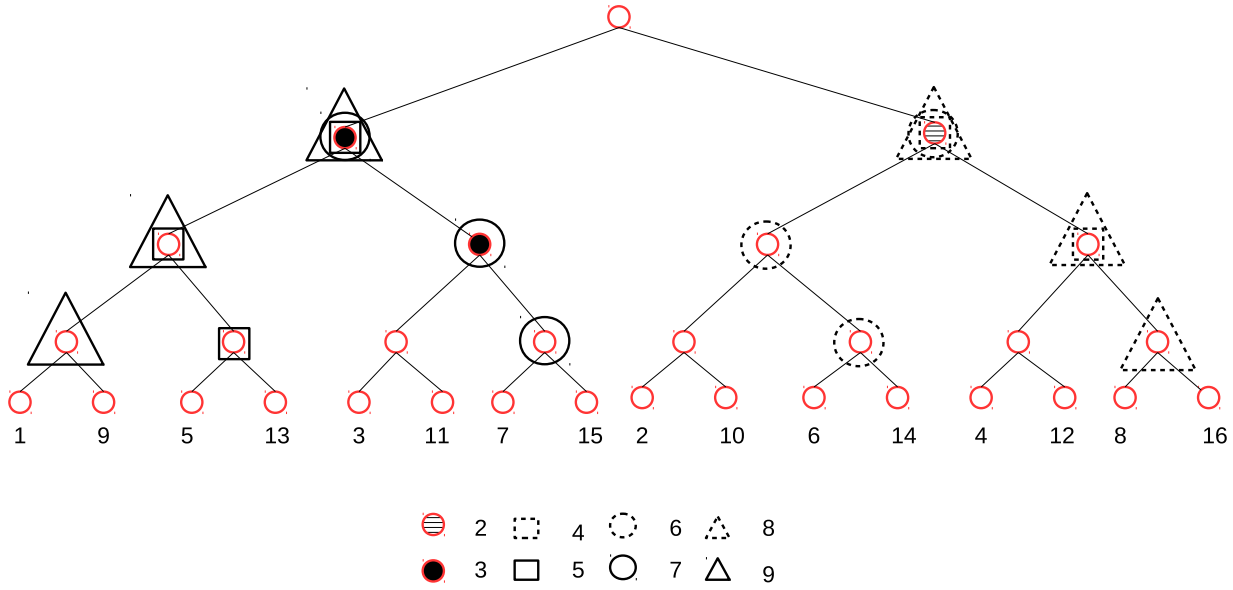

Figure 5.16: Illustration of nine evictions. Numbers below leaves represent the order of reverse deterministic lexicographic eviction. Buckets with same shapes were full and then evicted at the same step. Example: buckets with triangular shape are evicted in step 9.

originally empty, will be overwritten by a copy of the root. The root is finally overwritten by an empty bucket. The bucket $B_{1,j-2^0}$ will be merged with its destination child $B_{2,j-2^1}$ then emptied. The sibling of the bucket $B_{1,j-2^0}$ will be overwritten by the content of $B_{1,j-2^0}$. We reiterate the process until the end of the path (this was a recapitulation of Evict).

Given a bucket $B_{i,j}$, we are interested in enumerating the eviction's steps of creation of all buckets that have contributed to bucket $B_{i,j}$. The bucket composition also follows a pattern that is unique to any bucket in the construction. Given the eviction algorithm, every bucket in the $i$thlevel is created by merging all buckets in the path from the root to the $(i-1)^{\text{th}}$ level, see Table 5.4 for an example of this pattern for $N = 16$. As an example, the bucket in path 9 at the 3^{rd} level was created during the 5^{th} eviction step. To determine the buckets that contributed to this bucket's creation, we check the column that has an evicted path equal to 5. Then, we consider all buckets that are in upper levels: buckets 4 and 3 which are in levels 2 and 1. In general, a bucket $B_{i,j}$ is the result of merging the following buckets: $\{B_{0,j}, B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{i-1,j-2^{i-2}}\}$.

Noisy blocks It is important to understand the source of provenance of noisy blocks. From a one hand, a noisy block can be created whenever an access has been performed on C-ORAM.

Therefore, the accessed block is not valid anymore and should be turned to a noisy block by updating the headers. On the other hand, a noisy block can be also created from the eviction process. During an eviction, and in particular, when a parent is copied to its sibling, many real elements are no longer valid and become noisy. The main goal of this section is to upper bound the number of noisy blocks in all buckets with high probability. Our quantification has then to take into account both sources, however, one can show that the first source of noisy blocks can be included as a worst case of the second source. Recall that a bucket cannot hold more than z real elements which means that we can have up to z real elements turning to noise –if we access the same bucket z times before eviction–. One can only add z additional blocks to each bucket to handle reads, so whatever computed bound on ϕ, one can increase it by one. However, one can show that it is not necessary. In fact, this situation is equivalent to having all z real elements in a given bucket as noise for its sibling (which is the worst case). Thus, one can consider the first source of noisy blocks as a sub-case of the second one. We are now ready to state our main theorem.

Theorem 5.3.1. *If $\phi \in \Theta(1)$, the probability that $Z_{i+1,j} \geq R_{i,j}$ and $Z_{i,j} \geq R_{i+1,j}$ is in $O(z^{-z})$, for all $i \in [L]$ and $j \in \mathbb{N}$.*

Proof. Based on our assumption, we know that a path cannot handle more than z real elements with high probability. This implies that $\forall i \in \{0, \ldots, L\}$, we have $R_{i+1} + R_i \leq z$.

Here, for sake of clarity and without loss of any generality, we omit the eviction step j from notation just to minimize the burden of additional indexes. To show that $\forall i \in [L]$, $Z_{i+1} \geq R_i$ and $Z_i \geq R_{i+1}$, it is equivalent to show that $\tilde{N}_i \leq \phi \cdot z$:

$$\begin{aligned} R_{i+1} + R_i &\leq z \\ R_{i+1} + R_i + \tilde{N}_i + Z_i &\leq z + \tilde{N}_i + Z_i \\ R_{i+1} + \mu \cdot z &\leq z + \tilde{N}_i + Z_i \\ R_{i+1} &\leq (\tilde{N}_i - \phi \cdot z) + Z_i \end{aligned}$$

Therefore, it is sufficient to show that $\tilde{N}_i - \phi \cdot z \leq 0$ in order to proof that $\forall i \in [L]$, $Z_{i+1} \geq R_i$ and $Z_i \geq R_{i+1}$. It is clear that these inequalities hold for any eviction step $j \in [N]$.

Consequently, the proof boils down to show that the probability that $\tilde{N}_{i,j} > \phi \cdot z$ is negligible with very high probability.

Based on the bucket composition history, notice that the noisy elements in the bucket $B_{i,j}$ are exactly those that exist already in the bucket $B_{i-1,j-2^{i-2}}$, plus, all the real elements that will be evicted to the other child and therefore they are considered noisy elements for the bucket $B_{i,j}$. Thus, we have $\Pr(\tilde{N}_{i,j} > \phi \cdot z) = \Pr(\tilde{N}_{i-1,j-2^{i-2}} + R'_{i-1,j} > \phi \cdot z)$.

We have shown in the bucket composition history that $B_{i,j}$ is created by summing all the buckets $\{B_{0,j}, B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{i-1,j-2^{i-2}}\}$. The above equation can be then formulated more accurately such that $\Pr(\tilde{N}_{i,j} > \phi \cdot z) = \Pr(\max_i(\tilde{N}_{1,j-2^0}, \ldots, \tilde{N}_{i-1,j-2^{i-2}}) + R'_{i-1,j} > \phi \cdot z)$.

The equation can be understood as follows: the noise in bucket $B_{i,j}$ is the maximal amount of noise in any bucket in its history. Each bucket is created independently of the other ones in the evicted path. Therefore the quantity of noise in every bucket in the evicted path is independent of the other ones. We give more details below about the independence assumption. Since the noise is cumulative during the eviction, the bucket that has the maximum noise will represent the noise of the last bucket. Recall that based on Algorithm 2, the noisy blocks are added up. Also, to this quantity of noise, we add the sum of all real elements in the path that are no longer considered real elements in $B_{i,j}$ and therefore represent a new noise denoted by $R'_{i-1,j}$.

All buckets in an evicted path are *independent* of each others, i.e., the number of real elements, the number of noisy elements are independent of the the other buckets in the path. This holds since the real elements, the noise in any bucket is generated from distinct evictions. First, note that a bucket is created by merging all its ancestors. We have defined in the previous section the notion of *bucket composition history* that keeps track of each bucket's ancestor that contributed to its creation. As have been shown, the bucket ancestors emanate from different evictions' steps which is a consequence of the reverse deterministic lexicographic eviction. More importantly, each bucket in the evicted path has a different bucket composition history such that the intersection of all of them is *empty*. Formally, an evicted path, based on the *path composition history*, of the $j$thstep equals $\{B_{1,j-2^0}, B_{2,j-2^1}, \ldots, B_{L-1,j-2^{L-2}}\}$.

Consider a bucket and its parent in the evicted path for $i \in \{1, \ldots, L-1\}$, $B_{i,j-2^{i-1}}$ and $B_{i+1,j-2^i}$. The bucket composition is $\{B_{0,j-2^{i-1}}, B_{1,j-2^{i-1}-2^0}, B_{2,j-2^{i-1}-2^1}, \ldots, B_{i-1,j-2^{i-1}-2^{i-2}}\}$ and $\{B_{0,j-2^i}, B_{1,j-2^i-2^0}, B_{2,j-2^i-2^1}, \ldots, B_{i,j-2^i-2^{i-1}}\}$.

By replacing each bucket in the above sequence by its own bucket composition history and by iterating the process, we will converge to a state where each bucket is composed of root buckets that were instantiated at different evictions' steps. That is, no distinct buckets in the evicted path have a root in common. Thus, the number of real and noisy elements are independent from each other. Therefore,

$$\begin{aligned}\Pr(\tilde{N}_{i,j} > \phi \cdot z) &= 1 - \Pr(\max_i(\tilde{N}_{1,j-2^0}, \ldots, \tilde{N}_{i-1,j-2^{i-2}}) \\ &\quad + R'_{i-1,j} \leq \phi \cdot z) \\ &= 1 - \prod_{k=1}^{i-1} \Pr(\tilde{N}_{k,j-2^{k-1}} + R'_{i-1,j} \leq \phi \cdot z)\end{aligned}$$

We can reiterate the process of counting the noise until arriving to the root. The quantity of noise in the root is null. Then

$$\begin{aligned}\Pr(\tilde{N}_{i,j} > \phi \cdot z) &= 1 - \prod_{k=1}^{i-1}\prod_{l=1}^{k-1}\cdots\prod_{t=1}^{s-1} \Pr(\tilde{N}_{0,t} + R'_{0,t} + \\ &\quad R'_{1,s} + \ldots + R'_{i-1,j} \leq \phi \cdot z) \\ &= 1 - \prod_{k=1}^{i-1}\prod_{l=1}^{k-1}\cdots\prod_{t=1}^{s-1} \Pr(R'_{0,t} + R'_{1,s} + \\ &\quad \cdots + R'_{i-1,j} \leq \phi \cdot z)\end{aligned}$$

Recall that $R'_{i-1,j}$ represents the number of real elements in the bucket $B_{i-1,j}$ that will be considered as noise in the bucket $B_{i,j}$. Any bucket cannot have more than z elements with hight probability, denoting $\mathcal{R} = R'_{0,t} + R'_{1,s} + \ldots + R'_{i-1,j}$, we then have

$$\begin{aligned}\Pr(\mathcal{R} \leq \phi \cdot z) &= 1 - \Pr(i \cdot z \geq \mathcal{R} \geq \phi \cdot z) \\ &= 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \Pr(\mathcal{R} = k) \qquad (5.21)\end{aligned}$$

Now, we have to compute an upper bound of $\Pr(\mathcal{R} = k)$. One can proceed by: (1) counting all possible solutions of $\mathcal{R} = k$, then (2) multiply this value by the probability of the most likely

solution. All the possible combinations of the equation $x_1 + \ldots + x_k = N$ equal $\binom{k+N-1}{N}$ possibilities. The non-trivial part is to find an upper bound of the most likely solution that, in its general form, equals $\Pr(R'_{0,t} = x_1 \text{ AND } \ldots \text{ AND } R'_{i-1,j} = x_{i-1})$. We have that R'_{i-1,x_i} follows a binomial distribution such that $\Pr(R'_{i-1,t} = x_i) \leq \binom{2^{i-1}}{x_i} \cdot \frac{1}{(2^i)^{x_i}}$. Using the independence between buckets, see above for the independence argument, we obtain:

$$\begin{aligned}\Pr(\bigwedge_{j=0}^{i-1} \Pr(R'_j = x_j)) &= \prod_{j=0}^{i-1} \Pr(R'_j = x_j) \leq \prod_{j=0}^{i-1} \binom{2^{j-1}}{x_j} \cdot \frac{1}{(2^j)^{x_j}} \\ &\leq \prod_{j=0}^{i-1} (\frac{e}{2x_j})^{x_j} = (\frac{e}{2})^k \prod_{j=0}^{i-1} (\frac{1}{x_j})^{x_j}\end{aligned}$$

We want to find a readable upper bound only in function of k. Also, remark that the above inequality is true iff, $\forall j \in \{1, \ldots, i-1\}$, the following statement holds $x_j \leq 2^{j-1}$. One can verify with induction that with x_j's reaching their upper bounds 2^{j-1} minimizes $\prod_{j=1}^{i-1} x_j^{x_j}$ and therefore maximizes the inverse of the function. Also, there exists by construction a nonnegative integer γ such that $\sum_{i=1}^{\gamma-1} 2^i \leq k \leq \sum_{i=1}^{\gamma} 2^i$, which implies that $\gamma - 1 \leq \log k \leq \gamma + 1$. Putting everything together we obtain:

$$\begin{aligned}\Pr(\bigwedge_{j=0}^{i-1} \Pr(R'_j = x_j) &= (\frac{e}{2})^k \prod_{j=0}^{i-1} (\frac{1}{x_j})^{x_j} \\ &\leq (\frac{e}{2})^k \cdot \frac{1}{2^2 \cdot (2^2)^{2^2} \cdots (2^{\gamma-1})^{2^{\gamma-1}}} \\ &= (\frac{e}{2})^k \cdot \frac{1}{2^{\sum_{j=1}^{\gamma-1} j 2^j}} = (\frac{e}{2})^k \cdot \frac{1}{2^{1+(\gamma-2)2^\gamma}} \\ &\leq (\frac{e}{2})^k \cdot \frac{1}{2^{1+(\log k - 3)k}} \leq (\frac{4e}{k})^k\end{aligned}$$

Now, we plug the above results in (5.21)

$$\begin{aligned}\Pr(\mathcal{R} \leq \phi \cdot z) &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} \binom{k+i-1}{k} (\frac{4e}{k})^k \\ &\geq 1 - \sum_{k=\phi \cdot z}^{i \cdot z} (\frac{4e^2 \cdot (k+i-1)}{k^2})^k \\ &\geq 1 - (i-\phi) \cdot z \cdot (\frac{8e^2 \cdot (\phi \cdot z + i - 1)}{2(\phi \cdot z)^2})^{\phi \cdot z} \qquad (5.22) \\ &\geq 1 - i \cdot z \cdot (\frac{8e^2}{\phi \cdot z})^{\phi \cdot z} \qquad (5.23)\end{aligned}$$

Inequalities (5.22) and (5.23) are bounds that are reached first by replacing $k = \phi \cdot z$ since it will result on the larger value (k is in the denominator) and by summing over the final probability by $i \cdot z$. Combining all results together, we have

$$\begin{aligned}
\Pr(\tilde{N}_{j,k} > \phi \cdot z) &\leq 1 - \prod_{k=1}^{i-1}\prod_{l=1}^{k-1}\cdots\prod_{t=1}^{s-1}(1 - \Pr(j \cdot z \geq \mathcal{R} \geq \phi \cdot z)) \\
&\leq 1 - (1 - \Pr(j \cdot z \geq \mathcal{R} \geq \phi \cdot z))^{O(\frac{i^i}{i!})} \\
&\leq 1 - (1 - i \cdot z \cdot (\frac{8e^2}{\phi \cdot z})^{\phi \cdot z})^{O(\frac{i^i}{i!})} \\
&= O(\frac{i^i}{i!} iz(\frac{e^2}{\phi \cdot z})^{\phi \cdot z}) = O(e^i iz(\frac{e^2}{\phi \cdot z})^{\phi \cdot z})
\end{aligned}$$

The last transitions are obtained by the binomial inequality and Stirling approximation. Now, we define the value of ϕ for which this probability is negligible. The probability above can be simplified to $\Pr(\tilde{N}_{i,j} > \phi \cdot z) = O(e^{i+\ln(i \cdot z)+2\phi \cdot z-\ln(\phi \cdot z)\cdot\phi \cdot z})$.

This probability computation is independent of the step of eviction $j \in \mathbb{N}$. Therefore, choosing $\phi \in \Theta(1)$ (and assuming that $L \in O(z \ln z)$), the probability equals: $\Pr(\tilde{N}_{i,j} > \phi \cdot z) \in O(e^{-z \ln z})$, which is negligible in z.

■

Corollary 5.3.1. *If bucket size $z \in \omega(\log N)$, $L \in \Theta(\log N)$, and $\phi \in \Theta(1)$, the probability that $Z_{i+1,j} \geq R_{i,j}$ and $Z_{i,j} \geq R_{i+1,j}$ is in $O(N^{-\log\log N})$, for all $i \in [L]$ and $j \in \mathbb{N}$.*

The Corollary can be derived from the main theorem by taking $z \in \omega(\log N)$.

5.3.4.2 Security Analysis

Oblivious merging We prove that permutations generated by Algorithm 2 are indistinguishable from random permutations. Informally, we show that the adversary cannot gain any knowledge about the load of a particular bucket. Applying a permutation from Algorithm 2 is equal to applying any randomly chosen permutation. We formalize our intuition in the security definition below.

First, we introduce our adversarial permutation indistinguishability experiment that we denote PermG. Let $\mathcal{M}$ denote a probabilistic algorithm that generates permutations based on the configurations of two buckets, and $\mathcal{A}$ a PPT adversary. Let k be the bucket size and s the security parame-

ter. By Perm we denote the set of all possible permutations of size k. Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and $\mathsf{SKE}^h = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ respectively denote an IND$-CPA encryption[2] and an IND-CPA additively homomorphic encryption schemes. $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s)$ refers to the instantiation of the experiments by algorithm $\mathcal{M}$, SKE, SKE^h and adversary $\mathcal{A}$.

The experiment $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s)$ consists of:

- Generate two keys K_1 and K_2 such that $K_1 \xleftarrow{\$} \mathsf{SKE}^h\mathsf{Gen}(1^s)$ and $K_2 \xleftarrow{\$} \mathsf{SKE}.\mathsf{Gen}(1^s)$ and send n buckets additively homomorphic encrypted with $\mathsf{SKE}^h.\mathsf{Enc}(K_1, .)$ associated to their headers encrypted with $\mathsf{SKE}.\mathsf{Enc}(K_2, .)$ to the adversary $\mathcal{A}$
- The adversary $\mathcal{A}$ picks two buckets A and B, then sends the encrypted headers $\mathsf{header}(A)$ and $\mathsf{header}(B)$
- A random bit $b \xleftarrow{\$} \{0, 1\}$ is chosen. If $b = 1$, $\pi_1 \xleftarrow{\$} \mathcal{M}(\mathsf{header}(A), \mathsf{header}(B))$, otherwise $\pi_0 \xleftarrow{\$} \mathsf{Perm}$. Send π_b to $\mathcal{A}$
- $\mathcal{A}$ has access to the oracle $\mathcal{O}_{\mathcal{M}}$ that issues permutation for any couple of headers different from those in the challenge
- $\mathcal{A}$ outputs a bit b'
- The output of the experiment is 1, if $b' = b$, and 0 otherwise. If $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s, b') = 1$, we say that $\mathcal{A}$ succeeded.

Definition 5.3.1 (Indistinguishable permutation)**.** *Algorithm $\mathcal{M}$ generates* indistinguishable permutations *iff for all PPT adversaries $\mathcal{A}$ and all possible configurations of buckets A and B, there exists a negligible function* negl*, such that*

$$|\Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s, 1) = 1] - \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s, 0) = 1]| \leq \mathsf{negl}(s).$$

Theorem 5.3.2. *If* SKE *is IND$-CPA secure,* SKE^h *IND-CPA secure, then Algorithm 2 generates indistinguishable permutations.*

[2]The adversary cannot distinguish between an encryption and a randomly generated bit string

Proof. We consider a sequence of games (Game_0, Game_1, Game_2) defined as follows:

Game_0 is exactly the experiment $\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s,1)$

Game_1 is similar to Game_0, except that encrypted headers are replaced with random strings

Game_2 is similar to Game_1, except that encrypted buckets are replaced with buckets with new randomly generated blocks which are additively encrypted

From the definition above, we have

$$\Pr[\mathsf{Game}_0] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s,1) = 1]. \tag{5.24}$$

For Game_1, we can construct an efficient distinguisher B_1 that reduces SKE to $\mathsf{IND\$\text{-}CPA}$ security such that

$$|\Pr[\mathsf{Game}_0] - \Pr[\mathsf{Game}_1]| \leq \mathsf{Adv}^{\mathsf{IND\$|CPA}}_{B_1,\mathsf{SKE}}(s). \tag{5.25}$$

Similarly for Game_1, we can build an efficient distinguisher B_2 that reduces the security of SKE^h to $\mathsf{IND\text{-}CPA}$ security such that

$$|\Pr[\mathsf{Game}_1] - \Pr[\mathsf{Game}_2]| \leq \mathsf{Adv}^{\mathsf{IND|CPA}}_{B_2,\mathsf{SKE}^h}(s). \tag{5.26}$$

We will no show that $\Pr[\mathsf{Game}_2] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s,0) = 1]$. That is, we need to show that the distribution of the output of algorithm $\mathcal{M}$ has a uniform distribution over the set Perm.

For sake of clarity, we assume that the number of noisy slots is zero in both buckets. Therefore, slots in A and B are either full or empty. We can easily extend the proof for the case where we have full, empty and noisy blocks.

For clarity, let X denote the discrete random variable that represents the permutation selected by the adversary and by $\mathsf{Load}_{\mathsf{i,j}}$ the event of $\mathsf{load}(A) = i$ and $\mathsf{load}(B) = j$. By $\mathsf{load}(A)$, we denote the number of real elements in bucket A. If $b = 0$, the adversary receives a permutation π_0 selected uniformly at random. It is clear that $\mathcal{A}$ cannot distinguish it from another uniformly generated random permutation. Note that in this case, for buckets with k slots, the probability that adversary selects a permutation from Perm uniformly at random equals $\frac{1}{|\mathsf{Perm}|} = \frac{1}{k!}$. Thus, $\Pr[X = \pi_0] = \frac{1}{k!}$.

If $b = 1$, the adversary receives π_1. We need to show that the permutations output by $\mathcal{M}$ are uniformly distributed.

$$\begin{aligned}\Pr(X = \pi_1) &= \sum_{i,j \in [n]} \Pr(X = \pi_1 \text{ and } \mathsf{Load_{i,j}}) \\ &= \sum_{i,j \in [n]} \Pr(X = \pi_1 \mid \mathsf{Load_{i,j}}) \cdot \Pr(\mathsf{Load_{i,j}})\end{aligned}$$

We compute the probability of selecting a permutation while the loads of buckets A and B are fixed to i and j. The number of possible configurations of valid permutations equals $\mathsf{Valid} = \binom{k}{i} \cdot \binom{k-i}{j}$. This represents the number of possible permutation from which the client can choose to generate a valid permutation. From the adversary view, it should take into consideration all possible configurations of blocks in both buckets A and B. The total number of permutations computes to $\mathsf{Total} = \binom{k}{i} \cdot \binom{k}{j} \cdot \binom{k-i}{j} \cdot j! \cdot (k-j)!$. The first two terms count the possible configurations of the loads in both buckets while the three last terms are for valid permutations for a fixed setting of load distribution in the buckets. The cardinality of possible configurations equals the number of possible combinations from which we can select j empty blocks from $k - i$, i.e., $\binom{k-i}{j}$. We then multiply this last value by the possible permutations of the $k-i$ full blocks and the j empty blocks that are respectively equal to $(k-j)!$ and $j!$. That is,

$$\begin{aligned}&\Pr(X = \pi_1 \mid \mathsf{Load_{i,j}}) = \frac{\mathsf{Valid}}{\mathsf{Total}} \\ &= \frac{\binom{k}{i} \cdot \binom{k-i}{j}}{\binom{k}{i} \cdot \binom{k}{j} \cdot \binom{k-i}{j} \cdot j! \cdot (k-j)!} = \frac{1}{\frac{k!}{j! \cdot (k-j)!} \cdot j! \cdot (k-j)!} = \frac{1}{k!}\end{aligned}$$

We insert the result of this equation in the previous one and obtain $\Pr(X = \pi_1) = \sum_{i,j \in [n]} \frac{1}{k!} \cdot \Pr(\mathsf{Load_{i,j}}) = \frac{1}{k!}$.

Thus for the adversary, permutations output by $\mathcal{M}$ are uniformly distributed, i.e.

$$\Pr[X = \pi_1] = \Pr[X = \pi_0] = \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathcal{E}_1,\mathcal{E}_2}(s, 0) = 1] \tag{5.27}$$

Combining Equations 5.24, 5.25, 5.26, and 5.27, we obtain

$$\begin{aligned}&\Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s, 1)] = \Pr[\mathsf{Game}_0] \\ &\leq \Pr[\mathsf{Game}_1] + \mathsf{Adv}^{\mathsf{IND\$|CPA}}_{B_1,\mathsf{SKE}}(s) \\ &\leq \Pr[\mathsf{Game}_2] + \mathsf{Adv}^{\mathsf{IND|CPA}}_{B_2,\mathsf{SKE}^h}(s) + \mathsf{Adv}^{\mathsf{IND\$|CPA}}_{B_1,\mathsf{SKE}}(s) \\ &\leq \Pr[\mathsf{PermG}^{\mathcal{A}}_{\mathcal{M},\mathsf{SKE},\mathsf{SKE}^h}(s, 0)] + \mathsf{Adv}^{\mathsf{IND|CPA}}_{B_2,\mathsf{SKE}^h}(s) + \mathsf{Adv}^{\mathsf{IND\$|CPA}}_{B_1,\mathsf{SKE}}(s).\end{aligned}$$

■

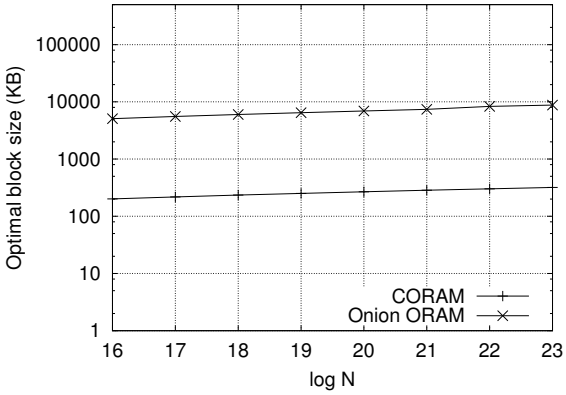

Figure 5.17: Minimum efficient block size for C-ORAM and Onion ORAM

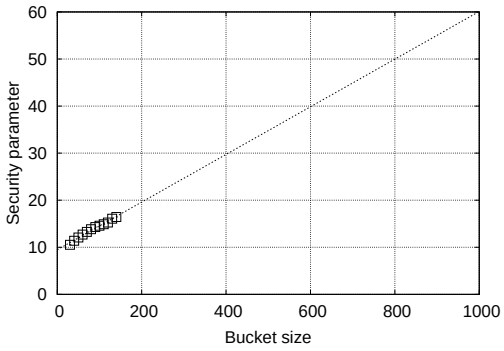

Figure 5.18: Required bucket size in relation to security parameter

Overflow probability of C-ORAM buckets C-ORAM eviction is similar to Onion ORAM [56]. The distribution of real elements for both constructions is exactly the same. We have a bucket size of $\mu \cdot z$ where z elements are allocated for real elements and $(\mu - 1)z$ is allocated for noisy elements to preserve the correctness of C-ORAM construction. The overflow probability denotes the fact that any bucket in C-ORAM will contain more than z elements. We want to show that this probability is negligible in n. For this, we borrow the results of [56] and [125] that have introduced the *eviction* factor χ. Throughout the paper, we have been stating that $\chi = O(z)$, which is a result of the following theorem, without explicitly stating it before to avoid confusion.

Theorem 5.3.3. *For the* eviction *factor χ and height L such that $z \geq \chi$ and $N \leq \chi \cdot 2^{L-1}$, the overflow probability after every eviction equals $e^{\frac{-(2z-\chi)^2}{6\chi}}$.*

Choosing $z \in \Theta(\lambda)$, $L \in \Theta(\log N)$, $\chi \in \Theta(\lambda)$ and $\lambda \in \omega(\log N)$ makes the the result of Th 5.3.3 negligible in N.

5.3.5 Evaluation

We have shown analytically that it suffices to set $\mu = \Theta(1)$ and have buckets of size $\Theta(z) = \Theta(\lambda)$. However, we have not derived precisely what bucket size is necessary for concrete security parameters. In order to get an idea of how bucket size in our scheme scales with λ, we performed a series of experiments simulating our ORAM and measuring the maximum number of used slots

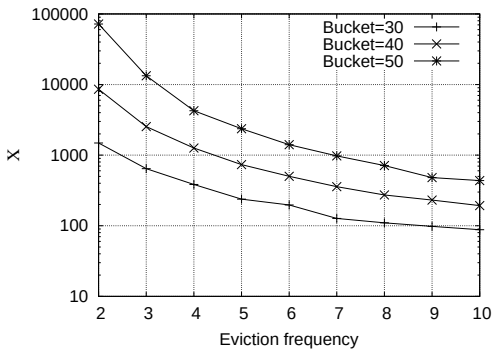

Figure 5.19: Required ciphertext operations for one access

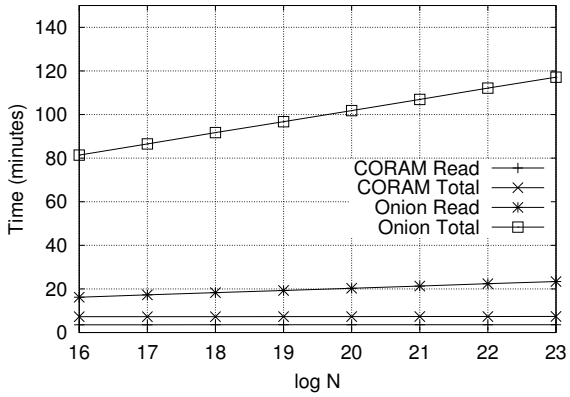

Figure 5.20: Comparison of computation time for one access

(real data blocks plus junk blocks) after x number of operations, for various values of x. We performed 20 sets of runs for each value of the security parameter where we executed 2^λ operations to test security parameter λ. For each of these runs, we measured the largest bucket load in the tree and then averaged this value across all runs to determine a bucket size which matches the selected security parameter. Figure 5.18 shows the results of this test, compared with equivalent tests run using the original Onion ORAM algorithm. Our results show that, because of the lower value of A in C-ORAM, our bucket size is actually slightly smaller than Onion ORAM.

Additionally, we compare the efficiency of our scheme in terms of server computation to that of Onion ORAM. We aim to quantify the number of homomorphic addition and multiplication operations in each scheme, to show that we have significant improvement. Throughout this analysis, we will consider a single multiplication or addition to be over an entire block, although in practice it may be divided into chunks of smaller ciphertext. Any changes in chunk size will apply equally to both schemes so discussion of its impact will be ignored. Note however that we do not have layered encryptions and so, in fact, ciphertext operations in our scheme will be cheaper simply because they are smaller.

During eviction Onion ORAM performs z select operations on each bucket, which each require a PIR query over z slots. This results in a total of z^2 multiplications for each bucket, over L buckets.

Amortized over z gives $O(z \cdot L)$ multiplications. Each multiplication also implies an addition in the select procedure, so the number of ciphertext additions is the same.

C-ORAM contains one major modification that is pertinent when comparing ciphertext operations: PIR queries are only done on the root bucket, to add new blocks, and on leaf buckets to read and remove blocks. C-ORAM then requires only $O(z \cdot \mu)$ multiplications and $z \cdot \mu \cdot L$ additions. Since we have shown that μ is a small constant, we effective gain a factor of $O(L)$ in multiplications. Crucially, this means the number of multiplications for C-ORAM is independent of $\log N$, as can be seen in Figure 5.19.

Figure 5.20 shows the results of tests we have run to determine the computational speedup of C-ORAM compared to Onion ORAM. We considered Pailler encryption as the homomorphic cipher for our tests, using a 2048-bit semiprime, which results in ciphertexts of size 4096 bits. Tests were done on a 2013 Macbook Pro with a 2.5 GHz Intel Core i7 processor, which we found could perform 62 ciphertext multiplications per second. We then calculated how much time it would take to perform the necessary ciphertext operations for one ORAM access, setting $B = 100kb$ and varying N from 2^{16} to 2^{23}. Figure 5.20 also shows the computation necessary for the online (read) portion of the access.

Although C-ORAM improves significantly over Onion ORAM, computation is the main bottleneck in both schemes. C-ORAM requires less than one MB of communication for one of the queries we tested. Using modern Internet connections, communication would take only a matter of seconds compared to a minute for the ciphertext computations. C-ORAM takes about 7 minutes for this query, while Onion ORAM takes over an hour and a half.

We stress that these evaluation results are largely to show the relative improvement of C-ORAM over Onion ORAM. We chose Paillier, because it is an established additively homomorphic encryption scheme, with well-understood levels of security. There are new homomorphic encryption schemes which perform much better than Paillier [32, 48, 96]. But stable, optimized implementations of them do not yet exist, and concrete parameters choices are still up for debate. Preliminary tests indicate that use of, for instance, the modern NTRU encryption scheme of [96] could allow for accesses with as little as 5 to 10 seconds of computation. However, a significant drawback that

must be balanced for NTRU is that the ciphertexts are much larger, resulting in a tradeoff between increased computational efficiency and higher communication for the PIR portions of C-ORAM. We leave full exploration of optimized homomorphic encryption schemes to future work.

Finally, we compare the optimal block size for C-ORAM in relation to Onion ORAM, cf. Figure 5.17. For each eviction, Onion ORAM requires $\lambda^2 L$ ciphertexts of size γ to be sent by the client, while we require only permutation vectors of total size $\mu\lambda L \log \lambda$. Since $\gamma = O(\lambda^3)$, this is a huge savings. For reads, Onion ORAM requires $\lambda L\gamma$ bits of ciphertext while we require only $4\mu\lambda\gamma$.

Comparison results: C-ORAM is able to achieve constant communication overhead in the worst-case, with significantly less server computation required in addition to smaller minimum block sizes. Figure 5.19 shows that we lower both the required number of ciphertext additions and multiplications by several orders of magnitude when compared to Onion ORAM, and Figure 5.20 shows that in practice this leads to a substantial improvement in efficiency. Figure 5.18 shows that, due to our lower value of A, the bucket size for C-ORAM is actually smaller in practice than Onion ORAM as well. Additionally, Figure 5.17 shows that C-ORAM requires much smaller blocks than Onion ORAM in practice.

Section Conclusion In this section, we presented C-ORAM, a solution to improve ORAM bandwidth. Constant ORAMs represent a first step towards making communication much cheaper at the cost of introducing computation overhead in the server side. Further works are required to carry this research direction forwards especially that many assumptions are still required.

Recently, Nayak et al. [115] demonstrate that our oblivious merge technique leak information when an adversary has an entire view of the tree structure. This will enables the adversary to correlate between permutations and track real blocks in the tree, therefore breaking the scheme's obliviousness. As a possible solution to fix this problem, one can think of finding out the correct blocks' parameters in the root in order to make this distinguisher fail with high probability. This will lead to re-define the correct number of noisy, real and empty blocks to put in the root before starting any eviction procedure. One needs then to prove that an adversary who had to see

all permutations in the tree cannot distinguish between these permutations, that is, the proof will not be based anymore on distinguishing between an oblivious merge permutation and a random permutation. As a consequence, this will increase the number of blocks in every bucket and therefore increase the number of bits in the headers. Another possible solution is to generate multiple PIR-Write vectors during the setup phase that will shuffle the buckets in such a way that any permutation outputted by the oblivious merge protocol will always operate on two freshly randomized buckets, but this has the disadvantage of requiring a costly setup phase, plus, inducing encrypted onions which will increase the block size and make the scheme much more inefficient and very similar to Onion ORAM [56].

Algorithm 6: GenPerm(A, B), oblivious permutation generation

Input: Configuration of buckets A and B
Output: A permutation randomly lining up bucket B to bucket A
// Slots in A and B start either empty, full or noisy; mark slots in A as *assigned* if block from B is assigned in π
Let x_1, x_2 be the number of empty and noisy slots in A;
Let y_1, y_2 be the number of full and noisy slots in B;
$d_1 = x_1 - y_1$;
$d_2 = x_2 - y_2$;
for i **from** *1* **to** $\mu \cdot z$ **do**
    case $B[i]$ *is full* **do** $z \xleftarrow{\$}$ all empty slots in A ;
    case $B[i]$ *is noisy* **do**
        if $d_2 > 0$ **then**
            $z \xleftarrow{\$}$ all noisy slots in A;
            $d_2 = d_2 - 1$;
        else
            $z \xleftarrow{\$}$ all empty slots in A;
        end
    end
    case $B[i]$ *is empty* **do**
        if $d_1 > 0$ **then**
            $z \xleftarrow{\$}$ all non-assigned slots in A;
            $d_1 = d_1 - 1$;
        else
            $z \xleftarrow{\$}$ all full slots in A;
        end
    end
    $\pi[i] = z$;
    $A[z] =$ assigned;
end
return π;

Algorithm 7: Access(op, adr, data, st): C-ORAM access operation, 2^{th} construction

Input: Operation op, address adr, data data, state st
Output: Block B associated to address adr

```
// Fetch tag value from position map
```
$\mathsf{tag} = \mathsf{posMap}(\mathsf{adr})$;
$\mathsf{posMap}(\mathsf{adr}) \xleftarrow{\$} [N]$;
```
// Retrieve desired block
```
$B =$Evict-Clone$(\mathsf{adr}, \mathsf{tag})$;
if $\mathsf{op} = \mathsf{write}$ **then** set $B = \mathsf{data}$;
```
// Select a random position in the root bucket
```
$\mathsf{pos}_1 \xleftarrow{\$} [\mu \cdot z]$;
```
// Write back the block to the empty root
```
PIR-Write$(\mathsf{pos}_1, B, \mathcal{P}(\mathsf{st}, 0))$;
Evict(st);
```
// Select a random noisy block position from the header of the leaf
    $\mathcal{P}(\mathsf{st}, L)$
```
$\mathsf{pos}_2 \xleftarrow{\$} \mathsf{header}^L$;
$N =$PIR-Read$(\mathsf{pos}_2, \mathcal{P}(\mathsf{st}, L))$;
```
// Write back the negation of the noisy block
```
PIR-Write$(\mathsf{pos}_2, -N, \mathcal{P}(\mathsf{st}, L))$;

Algorithm 8: Evict-Clone(adr, tag)

Input: Leaf tag and address adr
Output: Block B

Create a copy of the C-ORAM path $\mathcal{P}(\mathsf{tag})$;
for i **from** 0 **to** $L - 1$ **do**
    Retrieve header_1^i and header_1^{i+1};
    Retrieve C_i and C_{i+1} respectively the ith and the $(i+1)$th column of header_2^i and header_2^{i+1} of the bucket $\mathcal{P}(\mathsf{tag}, i)$ and $\mathcal{P}(\mathsf{tag}, i+1)$;
    // Generate the oblivious permutation π
    $\pi \leftarrow \mathsf{GenPerm}((\mathsf{header}_1^i, \mathsf{C}_i), (\mathsf{header}_1^{i+1}, \mathsf{C}_{i+1}))$;
    // Merge the parent and destination bucket
    $\mathcal{P}(\mathsf{tag}, i+1) = \pi(\mathcal{P}(\mathsf{tag}, i)) + \mathcal{P}(\mathsf{tag}, i+1)$;
end
$B =$PIR-Read$(\mathsf{adr}, \mathcal{P}(\mathsf{tag}, L))$;
for i **from** 0 **to** L **do**
    Update header_1^i in $\mathcal{P}(\mathsf{tag}, i)$;
end

Algorithm 9: PIR-Write(pos, block, $\mathcal{P}(\mathsf{tag}, \mathsf{level})$), PIR-write process

Input: Position pos, bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$, block B
Output: Updated bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$
`// Compute the` PIR-Write `vector` $\mathcal{V}$ `in client side`
for i **from** 1 **to** $\mu \cdot z$ **do**
  if $i \neq \mathsf{pos}$ **then** $\mathcal{V}_i \leftarrow \mathsf{SKE}^h.\mathsf{Enc}(0)$ **else**
  $\mathcal{V}_i \leftarrow \mathsf{SKE}^h.\mathsf{Enc}(1)$;
end
`// Write block in server side`
Parse bucket $\mathcal{P}(\mathsf{tag}, \mathsf{level})$ as $(\mu \cdot z \times |B|)$ binary matrix $\mathcal{M}$;
$\mathcal{M}_{i,j} = \mathcal{W}_i \cdot B_j$;
$\mathcal{P}(\mathsf{tag}, \mathsf{level}) = \mathcal{M} + \mathcal{P}(\mathsf{tag}, \mathsf{level})$;

Table 5.4: Bucket creation pattern in function of the eviction step.

Level_1	8	7	6	5	4	3	2	1	-
Level_2	7	6	5	4	3	2	1	-	-
Level_3	5	4	3	2	1	-	-	-	-
Evicted path	9	8	7	6	5	4	3	2	1

Chapter 6

Conclusion

> Work spares us from three evils: boredom, vice, and need.
>
> *Voltaire*

In this dissertation, we study the *encrypted search* problem and underline its considerable importance for end-users. We have introduced existing solutions, identified many challenges in the field, and proposed some possible solutions. Throughout this dissertation, we have been focusing on two main cryptographic primitives, symmetric searchable encryption (SSE) and its generalization to structured encryption (STE) and oblivious RAM (RAM) that both fulfill all of our main security and efficiency requirements. We have demonstrated that, nevertheless, there exist several inherent challenges to both STE and ORAM and proposed several contributions that enhance the state of the art with respect to: expressiveness and efficiency. First, we improved STE expressiveness by proposing boolean, semantic and substring STE constructions. Second, we have improved ORAM communication overhead by introducing a new recursive data structure that can replace traditional binary tree-based ORAM, we introduced the concept of resizability, and finally presented a shuffle-free solution that can eventually enable better communication saving.

Encrypted search is a hot topic and there are still many dimensions to investigate. As our future works, we continue along the same line of research and plan to propose: (1) the most optimal boolean STE, (2) a secure solution for relational database encryption, (3) a substring search over encrypted data with an obliviousness property, and (4) new efficient ORAM constructions.

Searching over encrypted data is a field that was introduced sixteen years ago. Researchers have made amazing advances towards creating efficient, more secure constructions. We believe that SSE constructions, by now, have reached some degree of maturity that will drive for real life deployments. Many start-ups and companies started already using different versions of SSE and/or PPE and other techniques to enable (somehow) searching over encrypted data.

Finally, with a long-term vision aiming to make SSE constructions available for developpers and pratitioners, we have also started a new open source project, *Clusion* [83], that provides a new encrypted search framework implementing most recent efficient SSE constructions. We hope that this open source library will have a considerable impact on both academia and industry, with a hope that it will be, in the near future, a reference for the encrypted search field.

Bibliography

[1] Always encrypted. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[2] Always encrypted. https://cloud.google.com/bigquery/.

[3] Amazon web serivces. https://aws.amazon.com.

[4] Anthem data breach. https://en.wikipedia.org/wiki/Anthem_medical_data_breach.

[5] Box. https://www.box.com.

[6] Dropbox. http://www.dropbox.com.

[7] Glass cubes. http://www.glasscubes.com/Collaborative.

[8] Google drive. https://www.google.com/drive.

[9] Hp public cloud. http://www.hpcloud.com/console.

[10] Ibm cloud. http://www.ibm.com/cloud-computing.

[11] JSci - A Science API for Java. Available at http://jsci.sourceforge.net/, Last accessed 03/03/2016.

[12] Microsoft azure. http://azure.microsoft.com.

[13] Premera bluecross blueshield data breach. http://www.forbes.com/sites/katevinton/2015/03/17/11-million-customers-medical-and-financial-data-may-have-been-exposed-in-premera-blue-cross-breach.

[14] Seeed. https://www.sics.se/sites/default/files/pub/andreasschaad.pdf.

[15] Zimbra. https://www.zimbra.com.

[16] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. M. Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. In V. Shoup, editor, *Advances in Cryptology – CRYPTO '05*, volume 3621 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2005.

[17] Amazon. Amazon s3 pricing, 2014. http://aws.amazon.com/s3/pricing/.

[18] Anonymous. r-ORAM implementation, 2015. https://www.dropbox.com/s/1tv04fgdxqgnoys/ORAM%20implementations.zip?dl=0.

[19] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable Oblivious Storage. In *Proceedings of Public-Key Cryptography*, pages 131–148, Buenos Aires, Argentina, 2014.

[20] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Engineering security and performance with cipherbase. *IEEE Data Eng. Bull.*, 35(4):65–72, 2012.

[21] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1101–1114, 2016.

[22] F. Baldimtsi and O. Ohrimenko. Sorting and searching behind the curtain. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, pages 127–146, 2015.

[23] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the o(n1/(2k-1)) barrier for information-theoretic private information retrieval. In *IEEE Symposium on Foundations of Computer Science (FOCS '02)*, pages 261–270, 2002.

[24] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[25] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS 2008)*, pages 257–266. ACM, 2008.

[26] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[27] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO '11*, pages 578–595, 2011.

[28] A. Boldyreva, S. Fehr, and A. O'Neill. On notions of security for deterministic encryption, and efficient constructions without random oracles. In *Advances in Cryptology - CRYPTO '08*, pages 335–359. 2008.

[29] D. Boneh, G. di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT '04*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004.

[30] D. Boneh and B. Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *Proceedings of the 4th Theory of Cryptography Conference*, Amsterdam, The Netherlands, February 2007.

[31] R. Bost. Sophos - forward secure searchable encryption. *IACR Cryptology ePrint Archive*, 2016:728, 2016.

[32] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[33] K. Brown. Balls in bins with limited capacity, 2014. http://www.mathpages.com/home/kmath337.htm.

[34] CALO Project. Enron Email Dataset. Available at http://www.cs.cmu.edu/~enron/, Last accessed 03/03/2016.

[35] G. Casella and R. Berger. *Statistical inference*. Duxbury advanced series in statistics and decision sciences. Thomson Learning, 2002.

[36] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 668–679, 2015.

[37] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[38] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[39] D. Cash, F. Liu, A. O'Neill, and C. Zhang. Reducing the leakage in practical order-revealing encryption. *IACR Cryptology ePrint Archive*, 2016:661, 2016.

[40] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[41] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS '05)*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.

[42] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[43] M. Chase and S. Kamara. Structured encryption and controlled disclosure. *IACR Cryptology ePrint Archive*, 2011:10, 2011.

[44] M. Chase and E. Shen. Substring-searchable symmetric encryption. *PoPETs*, 2015(2):263–281, 2015.

[45] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, Nov. 1998.

[46] K.-M. Chung and R. Pass. A Simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.

[47] CNN. The Interview revenge hack cost Sony just $15m , 2016. `http://money.cnn.com/2016/09/22/technology/yahoo-data-breach/`.

[48] J.-S. Coron, T. Lepoint, and M. Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *Public-Key Cryptography–PKC 2014*, pages 311–328. Springer, 2014.

[49] G. D. Crescenzo and V. Saraswat. Public Key Encryption with Searchable Keywords Based on Jacobi Symbols. In *Proceedings of the 8th International Conference on Cryptology in India*, Chennai, India, December 2007.

[50] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[51] D. Dachman-Soled, C. Liu, C. Papamanthou, E. Shi, and U. Vishkin. Oblivious network RAM and leveraging parallelism to achieve obliviousness. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, pages 337–359, 2015.

[52] I. Damgard, S. Faust, and C. Hazay. Secure two-party computation with low communication. In *Theory of Cryptography (TCC '12)*, volume 7194 of *Lecture Notes in Computer Science*, pages 54–74. Springer, 2012.

[53] I. Damgard, M. Geisler, M. Krøigaard, and J.-B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Conference on Practice and Theory in Public Key Cryptography (PKC '09)*, pages 160–179. Springer, 2009.

[54] I. Damgard, Y. Ishai, M. Krøigaard, J.-B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology - CRYPTO 2008*, pages 241–261, 2008.

[55] I. Damgard, S. Meldgaard, and J.-B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography Conference (TCC '08)*, volume 6597, page 144, 2011.

[56] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. *IACR Cryptology ePrint Archive*, 2015:5, 2015.

[57] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 269–283, 2012.

[58] F. B. Durak, T. M. DuBuisson, and D. Cash. What else is revealed by order-revealing encryption? *IACR Cryptology ePrint Archive*, 2016:786, 2016.

[59] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 123–145, 2015.

[60] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: A scalable private DBMS. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 395–410.

[61] C. Fletcher, L. Ren, A. Kwon, M. van Dijk, E. Stefanov, and S. Devadas. RAW Path ORAM: A Low-Latency, Low-Area Hardware ORAM Controller with Integrity Verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[62] C. W. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.

[63] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: round-optimal oblivious RAM with applications to searchable encryption. *IACR Cryptology ePrint Archive*, 2015:1010, 2015.

[64] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[65] C. Gentry, K. Goldman, S. Halevi, C. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and Using It Efficiently for Secure Computation. In *Proceedings of Privacy Enhancing Technologies*, pages 1–18, 2013.

[66] E.-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See `http://eprint.iacr.org/2003/216`.

[67] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing –STOC*, pages 182–194, New York, USA, 1987.

[68] O. Goldreich. *The Foundations of Cryptography – Volume 2*. Cambridge University Press, 2004.

[69] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *ACM Symposium on the Theory of Computation (STOC '87)*, pages 218–229. ACM, 1987.

[70] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[71] M. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages and Programming (ICALP '11)*, pages 576–587, 2011.

[72] M. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*, pages 95–100, 2011.

[73] M. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Symposium on Discrete Algorithms –SODA*, pages 157–167, Kyoto, Japan, 2012.

[74] D. Gordon, J. Katz, V. Kolesnikov, T. Malkin, M. Raykova, and Y. Vahlis. Secure computation with sublinear amortized work. Technical Report 2011/482, IACR ePrint Cryptography Archive, 2011. `http://eprint.iacr.org/2011/482`.

[75] W. He, D. Akhawe, S. Jain, E. Shi, and D. X. Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1028–1039, 2014.

[76] J. Hsu and P. Burke. Behavior of tandem buffers with geometric input and markovian output. *Communications, IEEE Transactions on*, 24(3):358–361, 1976.

[77] Y. H. Hwang and P. J. Lee. Public Key Encryption with Conjunctive Keyword Search and Its Extension to a Multi-user System. In *Proceedings of the 1st International Conference in Pairing-Based Cryptography*, Tokyo, Japan, July 2007.

[78] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[79] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS '13)*, pages 875–888, 2013.

[80] Y. Jia, T. Moataz, S. Tople, and P. Saxena. Oblivp2p: An oblivious peer-to-peer content sharing system. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 945–962, 2016.

[81] S. Kamara. Course 2950-v, 2016.

[82] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sublinear complexity, 2016.

[83] S. Kamara and T. Moataz. Clusion, 2016.

[84] S. Kamara and T. Moataz. SQL on structurally-encrypted databases. *IACR Cryptology ePrint Archive*, 2016:453, 2016.

[85] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.

[86] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[87] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.

[88] J. Katz and L. Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *ACM Symposium on Theory of Computing (STOC '00)*, pages 80–86, 2000.

[89] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[90] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*, pages 143–156, 2012.

[91] E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science (FOCS '97)*, pages 364–373. IEEE Computer Society, 1997.

[92] I. Lazrig, T. Moataz, I. Ray, I. Ray, T. Ong, M. G. Kahn, F. Cuppens, and N. Cuppens-Boulahia. Privacy preserving record matching using automated semi-trusted broker. In *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*, pages 103–118, 2015.

[93] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[94] Y. Lindell, B. Pinkas, and N. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *Proceedings of the 6th international conference on Security and Cryptography for Networks (SCN '08)*, pages 2–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[95] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In *Proceedings of Information Security Conference*, pages 314–328, Singapore, 2005.

[96] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[97] J. B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.

[98] S. Lu and R. Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. In *TCC*, pages 377–396, 2013.

[99] P. Majumder, M. Mitra, S. K. Parui, G. Kole, P. Mitra, and K. Datta. Yass: Yet another suffix stripper. *ACM Trans. Inf. Syst.*, 25(4), 2007.

[100] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security Symposium*, pages 20–20. USENIX Association, 2004.

[101] T. Mayberry, E.-O. Blass, and A. Chan. Efficient Private File Retrieval by Combining ORAM and PIR. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2014.

[102] J. Mayfield and P. McNamee. Single n-gram stemming. pages 415–416, 2003.

[103] M. Melucci and N. Orio. A novel method for stemmer generation based on hidden markov models. pages 131–138, 2003.

[104] X. Meng, S. Kamara, K. Nissim, and G. Kollios. GRECS: graph encryption for approximate shortest distance queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 504–517, 2015.

[105] T. Moataz, E.-O. Blass, and G. Noubir. Recursive Trees for Practical ORAM. In *Proceedings of Privacy Enhancing Technologies Symposium*, pages 115–134, Philadelphia, USA, 2015.

[106] T. Moataz, N. Cuppens-Boulahia, F. Cuppens, I. Ray, and I. Ray. ELITE: zero links identity management system. In *Data and Applications Security and Privacy XXVIII - 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings*, pages 195–210, 2014.

[107] T. Moataz, B. Justus, I. Ray, N. Cuppens-Boulahia, F. Cuppens, and I. Ray. Privacy-preserving multiple keyword search on outsourced data in the clouds. In *Data and Ap-*

plications Security and Privacy XXVIII - 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings, pages 66–81, 2014.

[108] T. Moataz, T. Mayberry, and E.-O. Blass. Constant Communication ORAM with Small Blocksize. In *Proceedings of Conference on Computer and Communications Security*, pages 862–873, 2015.

[109] T. Moataz, T. Mayberry, E.-O. Blass, and A. Chan. Resizable Tree-Based Oblivious RAM. In *Proceedings of Financial Cryptography and Data Security*, pages 147–167, San Juan, Puerto Rico, 2015. ISBN 978-3-662-47853-0.

[110] T. Moataz and A. Shikfa. Boolean symmetric searchable encryption. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 265–276, 2013.

[111] T. Moataz, A. Shikfa, N. Cuppens-Boulahia, and F. Cuppens. Semantic search over encrypted data. In *20st International Conference on Telecommunications, ICT 2013, Casablanca, Morocco, May 6-8, 2013*, pages 1–5, 2013.

[112] M. Naveed. The fallacy of composition of oblivious RAM and searchable encryption. *IACR Cryptology ePrint Archive*, 2015:668, 2015.

[113] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 644–655, 2015.

[114] M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[115] K. Nayak, L. Ren, I. Abraham, and B. Pinkas. An Oblivious RAM with Sub-logarithmic Bandwidth Blowup , 2016. IACR Cryptology ePrint Archive 849.

[116] R. Ostrovsky and V. Shoup. Private information storage. In *ACM Symposium on Theory of Computing (STOC '97)*, pages 294–303, 1997.

[117] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, and S. Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 359–374, 2014.

[118] V. Pappas, M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Private Search in the Real World. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011.

[119] D. J. Park, K. Kim, and P. J. Lee. Public Key Encryption with Conjunctive Field Keyword Search. In *Proceedings of the 5th International Workshop in Information Security Applications*, Jeju Island, Korea, August 2004.

[120] F. Peng, N. Ahmed, X. Li, and Y. Lu. Context sensitive stemming for web search. pages 639–646, 2007.

[121] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Advances in Cryptology – CRYPTO*, pages 502–519, Santa Barbara, USA, 2010.

[122] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 85–100, 2011.

[123] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *To appear in the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[124] M. Raykova, B. Vo, T. Malkin, and S. M. Bellovin. Secure Anonymous Database Search. In *Proceedings of the first ACM Cloud Computing Security Workshop*, Chicago, Illinois, USA, November 2009.

[125] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants Count: Practical Improvements to Oblivious RAM , 2014. IACR Cryptology ePrint Archive 997.

[126] S. E. Robertson, C. J. van Rijsbergen, and M. F. Porter. Probabilistic models of indexing and searching. pages 35–56, 1980.

[127] E.-K. Ryu and T. Takagi. Efficient conjunctive keyword-searchable encryption. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications*, pages 409–414, Niagara Falls, Canada, May 2007.

[128] E. Shi, J. Bethencourt, H. T. H. Chan, D. X. Song, and A. Perrig. Multi-Dimensional Range Query over Encrypted Data. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, pages 350–364, Berkley, California, USA, May 2007.

[129] E. Shi, T.-H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)¡sup¿3¡/sup¿) worst-case cost. In *Advances in Cryptology - ASIACRYPT '11*, pages 197–214. Springer-Verlag, 2011.

[130] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[131] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS, 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[132] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 247–258, 2013.

[133] E. Stefanov and E. Shi. Oblivistore: High performance oblivious distributed cloud data store. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[134] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS '12*, 2012.

[135] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *ACM Conference on Computer and Communications Security (CCS '13)*, 2013.

[136] M. Strizhov and I. Ray. Multi-keyword similarity search over encrypted cloud data. In *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings*, pages 52–65, 2014.

[137] P. Wang, H. Wang, and J. Pieprzyk. Keyword Field-Free Conjunctive Keyword Searches on Encrypted Data and Extension for Dynamic Groups. In *Proceedings of the 7th International Conference, Cryptology and Network Security*, Hong-Kong, China, December 2008.

[138] P. Williams and R. Sion. Usable pir. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, USA, 2008.

[139] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security (CCS '08)*, pages 139–148, 2008.

[140] P. Williams, R. Sion, and A. Tomescu. PrivateFS: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 977–988. ACM, 2012.

[141] J. Xu and W. B. Croft. Corpus-based stemming using cooccurrence of word cariants. *ACM Trans. Inf. Syst.*, 16(1):61–81, 1998.

[142] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 707–720, 2016.