ABSTRACT[1]

Title of Dissertation: IMPROVING EFFICIENCY, EXPRESSIVENESS AND SECURITY OF SEARCHABLE ENCRYPTION

Ioannis Demertzis
Doctor of Philosophy, 2020

Dissertation Directed by: Professor Charalampos Papamanthou
Department of Electrical and
Computer Engineering

A large part of our personal data, ranging from medical and financial records to our social activity, is stored online in cloud servers. Frequent data breaches threaten to expose these data to malicious third parties, often with catastrophic consequences (estimated to several billion of US dollars annually). In this thesis, we use, extend and improve Searchable Encryption (SE) in order to build the next generation encrypted databases/systems that will prevent such undesirable situations. Our goal is to build systems that are both practical and provably secure, while allowing expressive search and computation on encrypted data. Towards this goal, we have proposed new SE schemes that achieve the following: (i) have better search/computation time, (ii) allow expressive queries such as range, join, group-by, as well as dynamic query workloads, and (iii) provide new adjustable security-efficiency trade-offs—leading to robust and efficient schemes even against very powerful adversaries.

[1]This thesis was awarded the Distinguished Dissertation Award by the Department of Electrical and Computer Engineering, University of Maryland (UMD)

IMPROVING EFFICIENCY, EXPRESSIVENESS AND SECURITY OF SEARCHABLE ENCRYPTION

by

Ioannis Demertzis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Charalampos Papamanthou, Chair/Advisor
Professor Daniel Abadi, Dean's Representative
Professor Dana (Glasner) Dachman-Soled
Professor Tudor Dumitras
Professor Elaine Shi

Dedication

To the ones who love me the most: My Parents - My Brother - My Sofia.

Acknowledgments

First and foremost, I am grateful to my advisor, Babis Papamanthou, for his continuous support, generosity and encouragement throughout the 5 years of my PhD studies. His ethos and mentorship exceeded all the expectations that I had, when joining the PhD program. I will be forever grateful to Babis for generously providing me with unlimited opportunities and liberty to explore my research passions. In particular, he allowed me to pursue research problems that I was personally very interested in, he always encouraged me to set and meet higher goals, he worked closely with me on hard research problems/questions, and most importantly he believed in me and my abilities. I will never forget that during our meetings, I always admired his deep research knowledge, his persistent love for formalism, and particularly enjoyed our passionate research discussions, which helped me rapidly grow as a researcher. Without Babis, none of this work would have been possible and I am truly honored to be among his first "academic children".

I would also like to thank the members of my committee, Prof. Daniel Abadi, Prof. Dana (Glasner) Dachman-Soled, Prof. Tudor Dumitras and Prof. Elaine Shi for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing my dissertation.

Special thanks to Dimitris Papadopoulos who has been like a co-advisor to me, not only because we collaborated for a significant part of my thesis work, but also

because he provided exceptional advice on several professional and personal matters. His help and friendship over the years have been invaluable. I would sincerely like to thank both, Dimitris and his wife Eirini for their friendship, support and hospitality, especially during my visit in Hong Kong.

Back in the Techical University of Crete, I would like to express my honest gratitude to my diploma thesis and master thesis advisor, Prof. Minos Garofalakis, for trusting me to begin conducting research on the area of encrypted databases, guiding my early research steps, and providing me with career advices in critical moments of my graduate studies. I would also like to thank Prof. Stavros Christodoulakis for the motivation, knowledge and encouragement he instilled in me for the area of Databases, and not only. I consider him to have been a true inspiration and mentor during my undergraduate studies.

I would also like to thank all my co-authors, namely, JG Chamani, A. Deligiannakis, S. Papadopoulos, Od. Papapetrou, S. Shintre, R. Talapatra. Each collaboration helped me further shape my perspective on research. I also want to thank my mentors at VISA Research, Shashank Agrawal and Payman Mohassel; my mentor at Symantec Research Labs, Saurabh Shintre; my mentors at MSR Melissa Chase and Esha Ghosh.

This journey has been especially fun thanks to my close friends Ioakeim, Nikos K., Kostas X., Manolis, Nikos P., Evripidis, Leda, Christos Z., Stavros, George K., Giorgos T., Christos M., Aria, Konstantinos S., Vaggelis, Spiros, Konstantinos M., Konstantinos K.; I would like to thank all of them for all the moments, surprises and experiences that we have shared together. Of course, throughout this academic

journey and particularly during my internships, conferences, visits to Boston, California, Seattle and Hong Kong, I had the opportunity to meet amazing people and make friends for life. I would like to apologize to those I have inadvertently left out and deeply thank all of them.

Last but not least, this thesis would not have been possible without the support, love and encouragement of my parents, Michalis and Chrysanthi, as well as my younger brother, Athanasios-Rafail, who stood by me and therefore I would like to thank them. Finally, I would like to thank my beloved Sofia for her love and support; as well as for always making things better even in the most challenging parts of this journey. This thesis is dedicated to them.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents vi

List of Tables ix

List of Figures x

Chapter 1: Introduction 1
- 1.1 Prior SE Schemes . . . 3
- 1.2 Contributions/Organization of the dissertation . . . 4

Chapter 2: Preliminaries 10
- 2.1 Negligible function . . . 10
- 2.2 Randomized Encryption (RND). . . . 10
- 2.3 Pseudorandom functions (PRF) . . . 11
- 2.4 Collision-Resistant Hash Function. . . . 11
- 2.5 Keyword Search vs. Database Search . . . 11
- 2.6 Searchable Encryption (SE) Definition . . . 13
- 2.7 Oblivious Primitives . . . 17
- 2.8 Compression Schemes . . . 18
- 2.9 Attacks on deterministically-encrypted systems. . . . 20

Chapter 3: Fast Searchable Encryption with Tunable Locality 21
- 3.1 Scheme with Optimal Locality . . . 24
- 3.2 Tuning the Locality of Our Scheme . . . 28
- 3.3 Optimizations of Our Scheme . . . 32
- 3.4 Security Analysis and Leakage Profile . . . 37
- 3.5 Dynamic SE (DSE) . . . 40
- 3.6 Experiments . . . 43
  - 3.6.1 Setup . . . 45
  - 3.6.2 In-memory Comparison . . . 48
  - 3.6.3 External Memory Comparison . . . 51
  - 3.6.4 Comparison with TwoChoiceAlloc . . . 56

Chapter 4: Efficient Searchable Encryption Through Compression 58
4.1 Supported Leakage Functions . . . 60
4.2 Our Approach . . . 61
4.2.1 Choosing Compression Algorithms . . . 63
4.2.2 Our SE Construction . . . 65
4.2.3 Our OSE Construction . . . 69
4.3 Experiments . . . 75
4.3.1 Setup . . . 75
4.3.2 microSE/microOSE Evaluation . . . 77

Chapter 5: Dynamic Searchable Encryption with Small Client Storage 82
5.1 Dynamic Searchable Encryption (DSE) . . . 89
5.2 From Static to Dynamic Schemes . . . 94
5.2.1 Amortized construction . . . 94
5.2.2 De-amortized construction . . . 100
5.3 Efficient DSE with Quasi-Optimal Search . . . 108
5.4 Experimental Evaluation . . . 120
5.4.1 Search performance . . . 124
5.4.2 Update performance . . . 127
5.4.3 Client storage . . . 129
5.4.4 Quasi-optimal search performance for variable deletion percentages . . . 130

Chapter 6: SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage 132
6.1 Other Relevant Works . . . 135
6.2 Encrypted Databases from Searchable Encryption & Attacks . . . 137
6.2.1 SE-based Point Queries . . . 137
6.2.2 SE-based Join Queries . . . 138
6.2.3 SE-based Range Queries . . . 140
6.3 SEAL: Adjustable SE & Derived Constructions . . . 144
6.3.1 Adjustable Oblivious RAM . . . 145
6.3.2 Adjustable Padding . . . 149
6.3.3 SEAL . . . 151
6.3.4 New Constructions for Point/Join Queries . . . 155
6.3.5 New Constructions for Range Queries . . . 156
6.4 Evaluation Against Attacks . . . 157
6.4.1 Attacker Model . . . 158
6.4.2 Experimental Setup . . . 159
6.4.3 Attacking POINT-ADJ-SE . . . 161
6.4.4 Attacking JOIN-ADJ-SE . . . 165
6.4.5 Attacking RANGE-SRC-SE . . . 166
6.4.6 Efficiency of Adjustable Constructions . . . 168
6.4.7 Setting Parameters α and x in Practice . . . 171

6.5 Challenges for Dynamic Databases 175

Chapter 7: Conclusions and Future Work 177
7.1 Conclusions 177
7.2 Future Work 178

Bibliography 181

List of Tables

3.1 Comparison of the most representative SE schemes. We denote with N the number of keyword-document pairs, with m the number of unique keywords and with w the size of the result of a keyword search query. Our schemes can be parameterized in terms of locality L. Our most practical scheme is achieved by setting $L = 1$, yielding read efficiency $O(N^{1/(s+1)})$ and space equal to $\Theta(N \cdot s)$. Note that for $L = N^{1/s}$ (which gives constant read efficiency), we can prove our scheme is secure using as leakage only the size of the access pattern (as used in previous works). *Assuming no keyword list has size more than $N^{1-1/\log\log N}$.***Assuming no keyword list has size more than $N^{1-1/o(\log\log\log N)}$.***For a keyword-list with size $n = N^{1-\epsilon(n)}$. 23

4.1 Different leakages in our constructions. 61

5.1 Comparison of existing forward-and-backward-private DSE with small client storage. N is an upper bound for total insertions, $|W|$ = #distinct keywords. For keyword w: a_w = #updates, i_w = #insertions, d_w = #deletions, n_w = #files containing w. RT is #roundtrips for retrieving $DB(w)$. BP stands for backward privacy type (the smaller, the better) and $am.$ for amortized efficiency. The $\tilde{O}$ notation hides polylogarithmic factors. WO stands for storing search/insertion counters for each w at an oblivious map. To minimize client storage, oblivious map stashes are stored at the server and downloaded every time. 84

6.1 Query Recovery Attack for Range Queries QR_{SR} = # Correctly Decrypted Queries /# Queries) 168

List of Figures

2.1 SE ideal-real security experiments. 12

3.1 Example for $N = 64$ and $s = 7$. When $s = 2$, our scheme with optimal locality stores only levels 6 and 3, mapping all the queries of levels $0, 1, 2$ to level 3 and all the queries of levels $4, 5$ to level 6. The worst case read efficiency is $N^{1/2} = 8$ occurring when we map a query of size 1 to level 3. 25
3.2 Example for $N = 64$, $s = 2$ and $L = 2$. Our scheme stores levels $0, 3, 6$. The red arrows depict the queries whose answers contain false positives but with optimal locality, while the blue arrows show queries with optimal read efficiency and constant locality. 30
3.3 KeyGen and Setup algorithms for our scheme with locality L, read efficiency $O(N^{1/s}/L)$ and space $\Theta(s \cdot N)$. 33
3.4 Token and Search algorithms for our scheme with locality L, read efficiency $O(N^{1/s}/L)$ and space $\Theta(s \cdot N)$. 34
3.5 Simulator algorithms SimSetup and SimSearch for scheme with $O(L)$ locality and $O(N^{1/s}/L)$ read efficiency. 41
3.6 Simulator SimSearch for scheme with $O(N^{1/s})$ locality and $O(1)$ read efficiency. 42
3.7 Index costs 45
3.8 Search costs 47
3.9 Search Time 50
3.10 External memory comparison for the real dataset 50
3.11 External memory comparison ($N = 2^{37} - 1$) 53
3.12 External memory comparison using parallelism ($N = 2^{47} - 1$) 54
3.13 External memory comparison ($N = 2^{47} - 1$) 56

4.1 Our scheme first compresses the keyword lists and then performs the partitioning. Note that the packed words need to be stored with a rank, so that the decompression can work correctly. 63
4.2 Our more efficient SE construction using any SE and a set $\mathcal{C}$ of compression algorithms as black-box. 66
4.3 Simulator algorithms SimSetup and SimSearch for SE (keyword search problem) 67

4.4 Our OSE-K construction and the simulator algorithms SimSetup and SimSearch using an Oblivious RAM and a set $\mathcal{C}$ of compression algorithms as black-box. . . . 71
4.5 Index costs . . . 77
4.6 Search costs - Crime Dataset (Location attribute) . . . 78
4.7 Search costs - Crime Dataset (Date attribute) . . . 80
4.8 Search costs - Enron Dataset . . . 80
4.9 Additional Experiments (Crime Dataset) . . . 81

5.1 Real and ideal experiments for the DSE scheme. . . . 91
5.2 SD_a : from static to dynamic (amortized version). These are the encrypted indexes after five consecutive insertions $1-5$. Inserting element 1 requires the creation of EDB_0 which will contain element 1. Inserting element 2 requires downloading EDB_0 (to obtain element 1), creating EDB_1 which will contain elements 1 and 2, and deleting EDB_0. Searching for a keyword w requires to search all the active (non-deleted) encrypted indexes and return to the client all the individual search results. . . . 95
5.3 SD_a: from static to dynamic (amortized version). . . . 97
5.4 Static searchable encryption PiBas [1]. . . . 99
5.5 SD_d : from PiBas to DSE (de-amortized version). These are the encrypted indexes after 7 consecutive insertions $1-7$. Each level i contains 3 searchable encrypted indexes (OLDEST, OLDER, OLD) and one index (NEW) that is used for merging and rebuilding into a single index the OLDEST and OLDER indexes of the previous level $i-1$. The update algorithm passes through all levels and moves one element using OMAP_i from level $i-1$ to level i. Searching for a keyword w requires to search all the OLDEST, OLDER and OLD encrypted indexes and return to the client all the individual search results. . . . 100
5.6 SD_d: from PiBas to DSE (de-amortized version). . . . 101
5.7 SD_d: from PiBas to DSE (de-amortized version). . . . 102
5.8 Update tree for QOS with maximum insertions $N = 8$. Nodes are labeled with $[1, 15]$ leafs-to-root and left-to-right. This is the tree state after five insertions 1-5, and three deletions for $1, 2, 4$. A subsequent search starts from the Best Range Cover of leafs $[1, 5] = (13, 5)$ and proceeds downwards until it finds a black node or a leaf. The result is $(3, 5)$. . . . 109
5.9 QOS: DSE with quasi-optimal search time $O(n_w \log(i_w))$. . . . 110
5.10 QOS: DSE with quasi-optimal search time $O(n_w \log(i_w))$. . . . 111
5.11 QOS: DSE with quasi-optimal search time $O(n_w \log(i_w))$. . . . 112
5.12 Synthetic Dataset—Search (a) computation time vs. variable result size for $|DB| = 1\text{M}$, (b) computation time vs. variable $|DB|$ for result size 100, (c) communication size vs. variable result size for $|DB| = 1\text{M}$. . . . 121

5.13 Crime Dataset—Search (a) computation time vs. variable result size, (b) communication size vs. variable result size. . . . 122
5.14 Synthetic Dataset—Update (a) computation time vs. variable $|DB|$, (b) communication size vs. variable $|DB|$, (c) computation time with SD_a for 1000 updates staring from empty DB. . . . 123
5.15 Synthetic Dataset—Search computation time for $|DB| = 1$M and variable deletion percentage for: (a) $i_w = 100$ using OMAP, (b) $i_w =$ 20K using OMAP, (c) $i_w = 100$ storing word counters locally, (d) $i_w = 20$K storing word counters locally. . . . 126
5.16 Crime dataset—Search computation time and variable deletion percentage for: (a) $i_w = 78$ using OMAP, (b) $i_w = 24698$ using OMAP, (c) $i_w = 78$ storing word counters locally, (d) $i_w = 24698$ storing word counters locally. . . . 127

6.1 LOGARITHMIC-SRC [2, 3] consists of a full binary tree over the domain with an extra internal node between every two cousins. Red values denote the number of tuples each node contains (used for the proposed attack). . . . 142
6.2 ADJ-ORAM-α real-ideal security experiments. With $m_0, m_1, \ldots$, we denote the messages exchanged at Line 5 of both experiments. . . . 146
6.3 ADJ-ORAM-α using any ORAM as a black box. . . . 147
6.4 ADJ-Padding-x leading to $\log_x N$ different sizes. . . . 150
6.5 Our SEAL(α, x) scheme using ADJ-ORAM-α, ADJ-PADDING-x, and an oblivious dictionary as black boxes. . . . 151
6.6 Query Recovery Attack for Point Queries. . . . 160
6.7 Database Recovery Attack for Point Queries. . . . 160
6.8 Query Recovery Attack against POINT-ADJ-SE for various x. . . . 163
6.9 Query Recovery Attack against POINT-ADJ-SE for various x. . . . 163
6.10 Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. We show all attributes. . . . 165
6.11 Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. Attributes 4,7,11,20. . . . 166
6.12 Database Recovery Attack for Foreign-key Join Queries for the TPC-H Benchmark. . . . 167
6.13 Slowdown from SE. . . . 169
6.14 Speedup from sequential scan. . . . 170
6.15 Index Costs - Crime Dataset . . . 170
6.16 Search costs - Crime Dataset (Attribute 5) . . . 171
6.17 Search costs - Crime Dataset (Attribute 8) . . . 172

Chapter 1: Introduction

Nowadays, a large part of our personal data, ranging from medical and financial records to our social activity, are stored online in cloud servers. Frequent data breaches threaten to expose these data to malicious third parties, often with severe consequences (estimated to several billion of US dollars annually). To prevent such undesirable situations policymakers impose strict regulations on companies storing user data, e.g., the recent GDPR European Union regulation requires that data are stored encrypted and claims that "*encrypted storage is relatively safe only if the data owner, not the cloud service, holds the decryption keys*". Thus, the following question arises: *how can one efficiently directly query and compute on encrypted data without first decrypting these data*, a subject that has been the focus of a rapidly growing line of research over the past few years.

Searchable Encryption (SE) enables a data owner to outsource a database to a server in a private manner so that the latter can still private search queries without decrypting the database. In a typical SE scheme, the data owner prepares an encrypted index which is sent to the server. To perform a search query, the data owner sends a token to the server that allows her to utilize the encrypted index and retrieve the encrypted results (answering the requested query). SE schemes have

been proven to be very practical at the expense of well-defined leakages. These leakages reveal the *search pattern* (whether a query q has been made in the past or not), and the *access pattern* that consists of the *volume pattern* (number of tuples contained in the query result) and the *overlapping pattern* (which tuples in the result of query q appeared in the result of a previous query).

Existing techniques can reduce the above leakages at the expense of significantly increasing the queries' computational cost, e.g., use of oblivious algorithms and oblivious RAM [4, 5]. Recently, Zheng et al. [6], Eskandarian et al. [7], and Priebe et al. [8] proposed oblivious relational encrypted databases combining *trusted hardware* with *oblivious algorithms* to minimize the leaked information to just the size of accessed tables. Note that trusted hardware alone [9, 10] is not sufficient because it leaks the memory access pattern if not paired with oblivious methods.

In contrast to the above cryptographic solutions with rigorous security guarantees, several schemes and encrypted database systems have been proposed (mainly in database venues) achieving the desired performance but now at the cost of more leakage. CryptDB [11] and Monomi [12] utilize *deterministic* and *order preserving encryption*[1] in order to support point/range queries and joins. These seminal works triggered a plethora of subsequent related works (e.g., [13, 14]), which were adopted later by commercial products, such as Google Big Query and Microsoft SQL 2016. The aforementioned works achieve very practical performance but it was recently shown that they are susceptible to various attacks [15]. For example the leaked

[1] *Deterministic encryption* leaks the distribution of the input data. *Order preserving encryption* leaks the distribution of the input data and their order.

statistical and order information from these systems allowed [15] to recover actual patient records in plaintext.

1.1 Prior SE Schemes

In 2000, Song et al. [16] presented the first SE scheme for private keyword search, secure under Chosen Plaintext Attacks (CPA)[2]. Goh [18] realized that CPA security is not sufficient for the case of SE schemes. Curtmola et al. [19] introduced the state-of-the-art security definitions for SE for both, non-adaptive settings, i.e., maintaining security only if all the queries are submitted at once in one batch, as well as adaptive settings, i.e., maintaining security even if the queries are progressively submitted, and provided constructions that satisfy these definitions. The work of Curtmola et al. [19] led the way for several new SE schemes [20, 21, 22, 23, 24, 25, 26, 27, 28], some of which allow updates [23, 24, 25, 27, 28], are parallelizable [25], reduce the I/O costs [29, 30, 31, 32] and the number of cryptographic operation during search [3], as well as extend SE to support more expressive queries, such as boolean, substring, wildcard, phrase, range, range aggregate, join, group-by and general SQL queries [2, 6, 7, 8, 26, 33, 34].

[2] A scheme is secure against Chosen Plaintext Attacks (CPA) if the ciphertexts do not reveal any information about the plaintext even if the adversary can observe the encryption of the messages of her choice. For a formal definition please see [17].

1.2 Contributions/Organization of the dissertation

The main challenge of establishing SE valuable for real-world applications is to provide solutions that are (i) provably secure and robust against known and unknown attacks, (ii) very efficient and ready for practical deployment, (iii) expressive enough to support the demands of modern applications, i.e., support various types of static and dynamic queries. In this thesis, we propose new SE schemes for **more efficient**, **more expressive** and **more secure** encrypted search.

Improving I/O Costs ([32]). Scalable and efficient SE schemes require that space, read efficiency and locality overheads are as low as possible. Towards this goal, in Chapter 3, we design and evaluate the first SE scheme with tunable locality/read efficiency and linear space, and despite from the fact that asymptotically it is worse than prior works, it is in practice more efficient than both in-memory and external-memory state-of-the-art SE schemes—**12**× and **577**×, respectively. Our construction can be tuned to achieve various trade-offs between space, read efficiency, locality, parallelism and communication overhead; an important feature for further optimizing its performance for generic memory architectures.

Reducing the Number of Cryptographic Operations ([35]). Previous SE schemes require from the server (or sometimes the client) to perform cryptographic operations to retrieve the result, the number of which is at least equal to the size of the query result. The main reason is that for security purposes a query result r is stored in $|r|$ random positions indexed by $|r|$ values, each of which is produced

by a cryptographic operation. SE schemes with optimal locality could potentially address this problem, but surprisingly current approaches with good locality, either increase the number of cryptographic operations (due to false positives), or reduce only the number of cryptographic operations performed by the server (e.g., [32]).

In Chapter 4, we take a more aggressive approach which aims at reducing the total number of cryptographic operations required by the entire search protocol. Our main idea is to utilize compression to securely reduce the size of the plaintext indexes before producing the encrypted searchable indices. Our solution can use any existing SE scheme as a black-box and any combination of lossless compression algorithms in order to improve the search performance of the underlying SE scheme. We experimentally demonstrate up to **188**× savings in search time for the keyword search problem. Combining our locality-aware schemes [31, 32] with [35] can lead to search time improvements of up to **3-4 orders of magnitude** compared with prior state-of-the-art SE schemes.

Dynamic SE with Small Client Storage ([36]). Recent research has focused on Dynamic SE (DSE) schemes that can efficiently support modifications in the encrypted dataset, without the need to re-initialize the protocol. From a security perspective, developing secure DSE schemes is challenging, due to the additional information that may be revealed to the server because of updates. Two relevant security notions have been proposed for DSE schemes, namely forward and backward privacy. Forward privacy ensures that a new update cannot be related to any previous operation (up until the related keyword is searched). Backward privacy

ensures that if a document containing keyword w is first deleted and then a search of w occurs, the result of this search does not reveal anything about the deleted document. Many DSE schemes have been recently proposed but the most efficient ones share the same limitation: they require maintaining an operation counter for each unique keyword, either locally stored on the client, or accessed obliviously on the server, during every operation.

In Chapter 5, we propose three new schemes that overcome the above limitation and achieve constant permanent client storage with improved search performance, both asymptotically and experimentally, compared to prior state-of-the-art works. Our first two schemes adopt a "static-to-dynamic" transformation which eliminates the need for oblivious accesses during searches. Therefore, they are the first practical schemes with minimal client storage and non-interactive search. Our third scheme is the first quasi-optimal forward-and-backward DSE scheme with only a logarithmic overhead for retrieving the query result (independently of previous deletions). While it does require an oblivious access during search to keep permanent client storage minimal, its practical performance is up to **4 orders of magnitude** faster than the best existing scheme with quasi-optimal search.

SEAL—SE with Adjustable Leakages ([37]). Recent attacks have exploited the access and search pattern leakages mentioned above to recover the plaintext database or the posed queries, casting doubt to the usefulness of SE to encrypted systems. In many cases, the practicality and applicability of these attacks are questionable, since these attacks (i) do not attack state-of-the-art schemes, (ii) assume

that the attacker knows a great percentage of the input distribution, (iii) assume that the query distribution is known to the attacker, or even (iv) assume that the input dataset has a specific structure, e.g., every tuple of a specific numeric attribute has a unique value and all the numeric values appear exactly once.

In Chapter 6, we operate under the assumption that all these attacks provide an actual threat and we propose effective and efficient mitigation techniques against them. We propose **SEAL**, a new family of **SE** schemes with **A**djustable **L**eakages, in which the amount of privacy loss is expressed in leaked bits of search or access patterns and can be defined at setup. Our experiments show that protecting just a few bits of leakage (e.g., three to four bits of access pattern) is enough for existing and even new more aggressive attacks to fail, and also renders SEAL's query execution time practical for real-world applications (a little over one order of magnitude slow-down compared to traditional SE-based encrypted databases). For the construction of SEAL we developed two adjustable primitives that can be of independent interest, an adjustable ORAM and an oblivious adjustable padding algorithm. SEAL can be used for building efficient encrypted databases (supporting point, range, group-by and join queries) that are robust against all-powerful attacks that can serve as a benchmark for measuring the robustness of previous/future leakage-abuse attacks (i.e., showing the success rate of an attack as a function of the leaked information). Finally, we present a more efficient adjustable construction for ranges that reduces the access pattern leakage and the volume pattern leakage *implicitly* by modifying an existing construction [2] and not by using SEAL as a black-box.

Dissertation Organization. Chapter 2 introduces the necessary background. Chapters 3-6 (as we describe above) present our proposed approaches for more efficient, expressive and secure SE. Chapter 7 concludes the dissertation and provides future research directions.

Other Authored/Co-authored Works. In [31], we ask whether it is possible to build a SE scheme with: (i) linear space, (ii) constant locality, and (iii) sublogarithmic read efficiency. We answer this question in the affirmative by designing the first such SE scheme, strictly improving upon the previously best known scheme by Asharov et al. [30] (with logarithmic read efficiency).

In [2, 3], we extend SE to work for more expressive queries, such as private range and range aggregate queries. We construct new novel Range SE (RSE) schemes with realistic security/performance trade-offs. We reduce range search to multi-keyword search using range-covering techniques with tree-like indexes. We demonstrate that, given any secure SE scheme, the challenge boils down to (i) formulating leakages that arise from the index structure, and (ii) minimizing false positives incurred by some schemes under heavy data skew. In [3], we propose generic and specialized ways to provide locality-aware RSE schemes.

In [38], we propose GraphOS, a specialized graph database management system with *oblivious query processing*, i.e., the first system that minimizes the information leaked to the cloud server to just the number of nodes and edges in the graph. GraphOS achieves less leakage than all existing secure graph databases. Moreover, it improves the performance of the previous state-of-the-art scheme, both asymp-

totically and experimentally, e.g., for our tested scenarios it requires **10–1062**× less time to perform a BFS/DFS traversal and **9–719**× less time to compute the Minimum Spanning Tree. On the technical side, GraphOS is based on a combination of secure hardware, of novel specialized for trusted hardware oblivious RAM and oblivious data structure schemes that improve prior state-of-the-art approaches.

Chapter 2: Preliminaries

We denote by $\lambda \in \mathbb{N}$ a security parameter. PPT stands for probabilistic polynomial-time. We write $out \leftarrow \mathsf{Alg}(in)$ to indicate the output of an algorithm Alg and $(client_{out}, server_{out}) \leftrightarrow \mathsf{Prot}(client_{in}, server_{in})$ to indicate the execution of a protocol Prot between a client and a server.

2.1 Negligible function

A function ν: $\mathbb{N} \rightarrow \mathbb{R}$ is negligible in λ, denoted by $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $\nu(\lambda) < 1/p(\lambda)$.

2.2 Randomized Encryption (RND).

We refer to a randomized symmetric encryption scheme with three polynomial-time algorithms as RND, i.e., *RND = (Gen, Enc, Dec)*, such that *Gen* takes as input a security parameter λ and returns a secret key k, *Enc* takes as an input a secret key and a message and outputs a ciphertext and *Dec* takes as an input the secret key k and a ciphertext and outputs the message that was encrypted. An RND scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts do not reveal any information about the plaintext even if the adversary can observe the encryption of

the messages of his choice. For a formal definition see [17].

2.3 Pseudorandom functions (PRF)

Let $Gen(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function , and $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^{\ell'}$ be a pseudorandom function (PRF) family. F is a secure PRF family if for all PPT adversaries Adv,

$$|\Pr[K \leftarrow Gen(1^\lambda); \mathsf{Adv}^{F(K,\cdot)}(1^\lambda) = 1] - \Pr[\mathsf{Adv}^{R(\cdot)}(1^\lambda) = 1]| \leq v(\lambda),$$

where $R : \{0,1\}^\ell \to \{0,1\}^{\ell'}$ is a truly random function.

2.4 Collision-Resistant Hash Function.

H is a collision-resistant hash function if two inputs a and b have the same $H(a) = H(b)$ with negligible probability. For a formal definition see [17].

2.5 Keyword Search vs. Database Search

SE was originally meant for private file/keyword search, but in [2, 32] we realized that SE can also be used for database search, and in particular for point and range queries. Keyword search and database search are very similar problems assuming for simplicity that in the latter case we want to support queries on a single attribute. Then, we can map the notion of keywords to the notion of attribute values, and the notion of documents to the notion of tuples, which allows the utilization of SE for database search. The only difference between these two problems is the following: While in the keyword search problem, two different keywords can

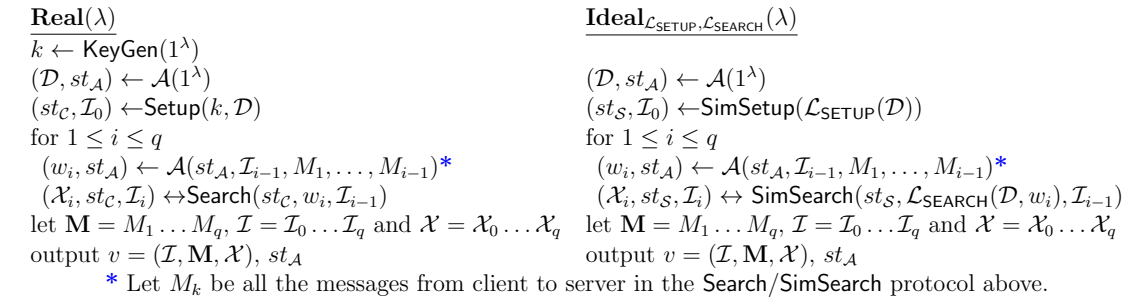


Figure 2.1: SE ideal-real security experiments.

map to the same document, in the database search problem, by definition of the problem, two different values of the same attribute will never map to the same tuple. For example, a patient cannot have more than one date of birth or SSN number. Therefore, the database search problem has less structural leakage.

Database search for multiple attributes. We can further extend SE to support private database search on multiple attributes. The first solution is to create m copies of the database, where m is the number of attributes, and use SE to encrypt each copy with a different key. This solution expands the space by a factor of m, but achieves optimal leakage, since it treats each attribute separately. The second solution considers only one copy of the database, but it increases the leakage. In particular, we create a single encrypted index for the values of all attributes by setting as searchable value v_i of attribute $attr_j$, the value $attr_j||v_i$. In this case a tuple id will be found in exactly m searchable values, leaking more information than before (e.g., the set of tuples matching queries on two different attributes).

2.6 Searchable Encryption (SE) Definition

Let $\mathcal{D}$ be a collection of *documents*. Each document $D \in \mathcal{D}$ is assigned a unique document identifier and contains a set of keywords from a dictionary Δ. We recall $\mathcal{D}(w)$ denotes the document identifiers of documents containing keyword w. SE schemes focus on building an *encrypted index* $\mathcal{I}$ on the document identifiers. For simplicity, we only consider the document identifiers instead of the actual documents since these are encrypted independently and stored in the server separately from the encrypted index $\mathcal{I}$; whenever the client retrieves a specific identifier during a search, he can send it to the server in an extra round and the server can send the corresponding documents back. Finally, N is the data collection size, i.e., $N = \sum_{\forall w \in \Delta} |\mathcal{D}(w)|$. A SE *protocol* considers two parties, a *client* and a *server* and consists of the following algorithms/protocols [19]:

- $k \leftarrow \mathsf{KeyGen}(1^\lambda)$: is a probabilistic algorithm performed by the client. It receives as input a security parameter λ and outputs a secret key k.
- $(st_\mathcal{C}, \mathcal{I}) \leftarrow \mathsf{Setup}(k, \mathcal{D})$: is a probabilistic algorithm performed by the client prior to sending any data to the server. It receives as input a secret key k and the data collection $\mathcal{D}$, and outputs an encrypted index $\mathcal{I}$. Index $\mathcal{I}$ is sent to the server. $st_\mathcal{C}$ is sent to the client and it contains only the secret key k.
- $(\mathcal{X}, st'_\mathcal{C}, \mathcal{I}') \leftrightarrow \mathsf{Search}(st_\mathcal{C}, w, \mathcal{I})$: is a protocol executed between the client and the server, where the client inserts the secret state $st_\mathcal{C}$ and a keyword w, while the server inserts an encrypted index $\mathcal{I}$. At the end of the protocol the

client learns $\mathcal{X}$, the set of all document identifiers $\mathcal{D}(w)$ corresponding to the keyword w and the updated secret state $st'_{\mathcal{C}}$, while the server's output is the updated encrypted index $\mathcal{I}'$.

We provide the security definition for the above SE scheme that corresponds to the real-ideal world paradigm [19], with a slightly modified syntax in order to match the security definition of OSE.

Definition 1 *Suppose* $(\mathsf{KeyGen}, \mathsf{Setup}, \mathsf{Search})$ *is a SE scheme based on the above definition, let* $\lambda \in \mathbb{N}$ *be the security parameter and consider experiments* $\mathbf{Real}(\lambda)$ *and* $\mathbf{Ideal}_{\mathcal{L}_{\mathsf{SETUP}},\mathcal{L}_{\mathsf{SEARCH}}}(\lambda)$ *presented in Figure 2.1, where* $\mathcal{L}_{\mathsf{SETUP}}$ *and* $\mathcal{L}_{\mathsf{SEARCH}}$ *are leakage functions to be defined next. We say that the SE scheme is* $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{SEARCH}})$*-secure*[1] *if for all polynomial-size adversaries* $\mathcal{A}$ *there exist polynomial-time simulators* $\mathsf{SimSetup}$ *and* $\mathsf{SimSearch}$*, such that for all polynomial time algorithms* Dist*:*

$$
\begin{aligned}
&|\Pr[\mathsf{Dist}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \mathbf{Real}(\lambda)] - \\
&\Pr[\mathsf{Dist}(v, st_{\mathcal{A}}) = 1 : (v, st_{\mathcal{A}}) \leftarrow \mathbf{Ideal}_{\mathcal{L}_{\mathsf{SETUP}},\mathcal{L}_{\mathsf{QUERY}}}(\lambda)]| \\
&\leq negl(\lambda)\,,
\end{aligned}
$$

where probabilities are taken over the coins of KeyGen *and* Setup *algorithms*[2]*.*

Figure 2.1 presents the real and ideal games for (semi-honest) adaptive adversaries, as introduced in [39]. These games are used to formally prove the security of

[1] In prior works $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{SEARCH}})$ is mentioned as $(\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_{\mathsf{SETUP}}$ or $\mathcal{L}_1$ refers to the total setup leakage, i.e. leakage prior to the query execution, and $\mathcal{L}_{\mathsf{SEARCH}}$ or $\mathcal{L}_2$ refers to the total query leakage, i.e. leakage during the query execution.

[2] A function ν: $\mathbb{N} \to \mathbb{N}$ is negligible in λ, $negl(\lambda)$, if for every positive polynomial $p(\cdot)$ and all sufficiently large λ, $\nu(\lambda) < 1/p(\lambda)$.

an SE scheme. They are partitioned into two worlds, the real and the ideal one. The real world represents a real SE scheme, where the adversary has access to the Setup and Search algorithms. More specifically, the real scheme creates a secret key to which the adversary does not have access. The adversary selects a document collection which is given as an input to the Setup algorithm. Furthermore, $st_{\mathcal{A}}$ denotes a state maintained by the adversary. The adversary observes the output of the Setup algorithm which is the encrypted index. Then, she selects a polynomial number of queries, and for each of these queries she observes the corresponding tokens. Having these tokens allows her to retrieve the encrypted result. In the ideal world, the adversary interacts with the simulator. The simulator $\mathcal{S}$, neither has access to the real document collection, nor to the real queries. Instead, the simulator only has access to predetermined leakage functions and by using these functions and her state she attempts to "fake" the algorithms Setup and Search. The adversary can only have access to one world, either to the real one, or to the ideal one. We consider only the strongest types of adversaries, i.e., adaptive adversaries that can select their own new queries based on previous ones. The adversary attempts to detect the world to which she has access. We prove that an adversary can distinguish the output of the real world from that of the ideal world only with negligible probability. This means that an adversary cannot learn anything more, than the predefined leakage.

As is common in SE definitions, we use two leakage functions, $\mathcal{L}_{\mathsf{SETUP}}$ and $\mathcal{L}_{\mathsf{SEARCH}}$. $\mathcal{L}_{\mathsf{SETUP}}$ is associated with what is leaked from the index alone, which means what is leaked prior to the query execution), whereas $\mathcal{L}_{\mathsf{SEARCH}}$ represents the

leakage produced by the queries (during the query execution). In particular

$$\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}) = N$$

is the *size pattern*, where $N = \sum_{\forall w \in \Delta} |\mathcal{D}(w)|$. Namely $\mathcal{L}_{\mathsf{SETUP}}$ leaks just the size of the index. Also

$$\mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w) = (id(w), \mathcal{D}(w))$$

is the *access pattern* leaking the identifiers of documents matching the query for keyword w, as well as a deterministic function of the keyword w, $id(w)$. The history of $\mathcal{L}_{\mathsf{SEARCH}}$ leakage also defines the *search pattern* leakage, which leaks whether two queries are the same.

As mentioned before, we can use an SE scheme for database search by corresponding the notion of document identifiers to tuple identifiers or encrypted tuples (encrypted using RND), and the notion of keywords to searchable attributes. In the case of database search, the $\mathcal{L}_{\mathsf{SETUP}}$ leakage is identical and corresponds to the number of tuples. The $\mathcal{L}_{\mathsf{SEARCH}}$ leakage in the database search differs from the previous case because it only contains the size of the encrypted results or similarly the size of the access pattern as shown below. We refer to this leakage as $\mathcal{L}^{DB}_{\mathsf{SEARCH}}$

$$\mathcal{L}^{DB}_{\mathsf{SEARCH}}(\mathcal{D}, w) = (id(w), |\mathcal{D}(w)|)$$

We consider the two cases separately because when performing database search the leakage is considered less significant than the keyword search problem. In particular,

SE in database search achieves leakage that is very close to the optimal one achieved by ORAMs. Their difference is that SE additionally leaks the search pattern.

The above security definition and leakages apply only to static SE schemes (in Chapter 5.1 we provide the security definition and leakages for Dynamic SE (DSE) schemes).

2.7 Oblivious Primitives

Oblivious RAM (ORAM). Oblivious RAM (ORAM), introduced in [40], is a compiler that encodes the memory such that accesses on the compiled memory do not reveal access patterns on the original memory. An ORAM scheme consists of two algorithms/protocols ORAM = (OramInitialize, OramAccess), where OramInitialize initializes the memory, and OramAccess performs the oblivious accesses. We provide the formal definition in Section 6.3.3.

Oblivious dictionary (ODICT). An oblivious dictionary is an oblivious data structure that can support oblivious queries from an arbitrary domain. ODict offers the following protocols (see [41] for a detailed description):

- $(T, \sigma) \leftarrow$ ODictSetup$(1^\lambda, N)$: Given a security parameter λ, and an upper bound N on the number of elements, it creates an oblivious data structure T. The client sends T to the server and maintains locally the state σ.

- $((\texttt{value}, \sigma'), T') \leftrightarrow$ ODictSearch$((\texttt{key}, \sigma), T)$: Given the search key `key` and σ, returns the corresponding value `value`, the updated T' and σ'.

- $(\sigma', T') \leftrightarrow$ ODictInsert$((\texttt{key}, \texttt{value}, \sigma), T)$: Given a key-value pair `key`, `value` and σ, it inserts this entry in the dictionary. It returns the updated T' and σ'.

In Chapter 5, we use in a black-box manner an *oblivious map (OMAP)* which is a privacy-preserving version of a key/value map data structure that aims to hide the type and content of a sequence of operations performed and can be implemented using an oblivious dictionary.

2.8 Compression Schemes

Our implementation uses (variations of) two different compression schemes, as we detail in the following.

FastPfor. FastPfor [42] is a modification of PforDelta [43]. Given a list of n integers, the algorithm begins by computing the deltas between two consecutive integers, and then it proceeds to compress the deltas. For example, let $I = \{2, 5, 10, 17\}$, then the deltas would be $I' = \{2, 3, 5, 7\}$ where $I'[0] = I[0]$ and $I'[i] = I[i] - I[i-1] (i \geqslant 0)$. The deltas are then split into chunks of 128 deltas and each of the chunks is compressed separately. For each chunk, the scheme chooses the smallest b, such that a majority of elements (controlled by a threshold, say 90%) can be encoded using b bits. The chunks are then stored using 128 b-bit locations, in addition to some extra storage for the values that could not be represented by the b bits (called *exceptions*). FastPfor enhances PforDelta because it stores the exceptions more efficiently.

EWAH. EWAH (Enhanced Word-Aligned Hybrid) [44] is a bitmap index compres-

sion algorithm, which is an enhanced modification of WAH (Word-Aligned Hybrid) [45]. Both algorithms belong to the RLE (Run Length Encoding) compression family. Given a set of n integers, we first create a bitmap in which we set the i-th element of the bitmap to 1 if and only if i is present in the list of input numbers. In WAH, the input bitmap is then split into groups of 31 bits. The groups are classified into two categories; if all the bits in a group are identical we consider it to be a filled group, otherwise a literal group. For example, 00000000000000000000000000000000 (0^{31} in short) is a filled group. Filled groups can be further classified into 0-fill groups (all bits are 0) and 1-fill groups (all bits are 1). WAH compresses a sequence of consecutive filled groups of the same type together using just one word. The scheme stores each literal group using one word (32 bits).

For instance, if the input bitmap is $10^{20}1^30^{111}1^{25}$ (160 bits), then WAH partitions it into 6 groups: G_1 ($10^{20}1^30^7$), G_2 (0^{31}), G_3 (0^{31}), G_4 (0^{31}), G_5 ($0^{11}1^{20}$) and G_6 ($0^{26}1^5$). Then, WAH encodes G_1 using ($010^{20}1^30^7$), i.e. the first bit is set to 0 denoting that it is a literal word and the remaining 31 bits contain G_1. Furthermore, it encodes G_2, G_3, G_4, G_5 together using ($100^{27}011$), i.e. the first bit is set to 1 indicating that it is a filled word, the second bit is set to 0 indicating that it is a 0-filled word and the remaining bits are used to store how many consecutive 0-groups are stored together. Finally, it encodes G_5 using ($00^{11}1^{20}$) and encodes G_6 using ($00^{26}1^5$).

EWAH is a modification of WAH because it addresses the latter's necessity to allocate too much space to store literal groups. Unlike WAH, EWAH divides an uncompressed input bitmap into 32-bit groups, whereas WAH uses 31-bit groups.

Then it encodes a sequence of p ($p \leqslant 65535$) fill groups and q ($q \leqslant 32767$) literal groups into a marker word followed by q literal words (stored in their original form). The first word in EWAH is always a marker word.

2.9 Attacks on deterministically-encrypted systems.

[15] proposed the *frequency analysis* and $\ell_{\mathbf{p}}$-*optimization* attacks that apply to databases encrypted with the use of deterministic schemes such as CryptDB [11].

The *frequency analysis* attack is the most basic and well-known inference attack in the area of cryptography. We define C_k and M_k to be the ciphertext and message spaces, respectively of the deterministic encryption scheme. Given a deterministically encrypted column $\mathbf{c}$ over C_k and an auxiliary dataset $\mathbf{z}$ over M_k, the attack works by assigning the i-th most frequent element of $\mathbf{c}$ to the i-th most frequent element of $\mathbf{z}$.

The $\ell_{\mathbf{p}}$-*optimization* attack is a family of attacks against deterministic encryption. The main goal is to find an assignment from ciphertexts to plaintexts that minimizes a given cost function, e.g., the $\ell_{\mathbf{p}}$ distance between the histograms of the dataset. This attack minimizes the total mismatch identified in frequencies across all plaintext and ciphertext pairs.

Chapter 3: Fast Searchable Encryption with Tunable Locality

Since the first work on SE was proposed in 2000 [16], most follow-up works considered scenarios where the encrypted index could fit in memory. However, for very large indexes and databases that must be stored on disk (e.g., see the CW-MC-OXT-4 dataset from the recent work of Cash et al. [1] whose encrypted index had size around 904 GB), these in-memory schemes cannot scale since random access is expensive. In these scenarios, the practical performance of SE schemes depends on the *locality*, namely the number of non-continuous locations that the server accesses for each query. Most SE schemes have poor locality (see the first five rows of Table 3.1), accessing one random location per result item—this random allocation of results in memory is necessary for achieving the desired security.

The work of Cash et al. [1] experimentally showed that in-memory SE cannot scale to large datasets, and therefore proposed new SE schemes with good locality guarantees. While trying to reduce[1] locality, it was observed that a number of additional entries per query must be read (usually referred to as false positives). The ratio of the total number of entries read over the size of the initial query result was

[1]Cash and Tessaro [29] and Asharov et al. [30] define locality as the number of non-continuous reads that the server makes per result item. Larger locality implies more non-continuous reads. Throughout this work we follow this notation and by reducing locality we mean improving locality, therefore $O(1)$ locality means optimal locality.

defined as *read efficiency*. Soon after, Cash and Tessaro [29] presented, along with a lower bound (see Table 3.1), a scheme that requires $\Theta(N \log N)$ space, $O(\log N)$ locality and optimal $O(1)$ read efficiency, where N is the number of document-keyword pairs in the document collection. Asharov et al. [30] presented three schemes: One with $\Theta(N \log N)$ space, optimal read efficiency and optimal locality ($N \log N$ scheme in Table 3.1) and two other schemes with linear space, optimal locality and very small (asymptotically) read efficiency (OneChoiceAlloc and TwoChoiceAlloc[2] in Table 3.1). Finally, Asharov et al. [50] further improved the efficiency of TwoChoiceAlloc by proposing a new scheme that works for bigger keyword-list sizes, i.e., up to $N^{1-1/o(\log\log\log N)}$, which has read efficiency that depends on the frequency of the queried keywords as shown in Table 3.1. Motivated by the above positive results and the impossibility result of [29], we propose in [31] a theoretical approach that asymptotically improves the best known scheme OneChoiceAlloc [30].

Among the above schemes, the $N \log N$ scheme is the fastest in practice (no false positives and no random access) and would be the scheme of choice if one could afford to store $\Theta(N \log N)$ space. For example, for the CW-MC-OXT-4 dataset [1] whose encrypted index has size approximately 904 GB and 2,732,311,945 entries, this would mean storing approximately 28.3 TB. In this thesis, we focus on SE schemes with good locality guarantees that occupy *linear space*. Such schemes are,

[2] TwoChoiceAlloc has very low read efficiency, i.e., $\Theta(\log\log N \log\log\log N)$ but is based on the assumption that all keyword lists in the dataset have size less than $N^{1-1/\log\log N}$. We tested this assumption for 4 real datasets: One containing crime records in Chicago since 2001 [46], the Enron email dataset [47], the USPS dataset [48] and the TPC-H dataset [49]. The Enron email dataset does not violate the assumption, which is not the case for the other datasets where almost half of the contained attributes violate it. For the crimes dataset, for example, the assumption was violated in 12 out of 21 attributes for 31% of the keywords on average.

Scheme	Locality	Read Efficiency	Storage
Kamara et al. [23]	$\Theta(w)$	$O(1)$	$\Theta(N+m)$
Curtmola et al. [19], Liesdonk et al. [21]	$\Theta(w)$	$O(1)$	$\Theta(N \cdot m)$
Kamara et al. [25]	$O(w \log N)$	$O(\log N)$	$\Theta(N \cdot m)$
Cash et al. [1] and [26]	$\Theta(w)$	$O(1)$	$\Theta(N)$
Stefanov et al. [24]	$O(w \log^3 N)$	$O(\log^3 N)$	$\Theta(N)$
Chase et al. [20]	$O(1)$	$O(1)$	$\Theta(N \cdot m)$
Cash et al. [29]	$O(\log N)$	$O(1)$	$\Theta(N \cdot \log N)$
Asharov et al. [30] ($N \log N$ scheme)	$O(1)$	$O(1)$	$\Theta(N \cdot \log N)$
Asharov et al. [30] (OneChoiceAlloc)	$O(1)$	$\Theta(\log N \log\log N)$	$\Theta(N)$
Asharov et al. [30] (TwoChoiceAlloc)*	$O(1)$	$\Theta(\log\log N \log\log\log N)$	$\Theta(N)$
Asharov et al. [50] (TwoChoiceAlloc+)**	$O(1)$	$\omega(1) \cdot \epsilon^{-1}(n) + O(\log\log\log N)$***	$\Theta(N)$
Demertzis et al. with $O(1)$ locality [32]	$O(1)$	$O(N^{1/(s+1)})$	$\Theta(N \cdot s)$
Demertzis et al. with $O(L)$ locality [32]	$O(L)$	$O(N^{1/s}/L)$	$\Theta(N \cdot s)$
Demertzis et al. [31]	$O(1)$	$O(\log^\gamma N)$, for $\gamma = \frac{2}{3} + \delta$ and $\delta > 0$	$\Theta(N)$
Lower bound [29]	$O(1)$	$O(1)$	$\omega(N)$

Table 3.1: Comparison of the most representative SE schemes. We denote with N the number of keyword-document pairs, with m the number of unique keywords and with w the size of the result of a keyword search query. Our schemes can be parameterized in terms of locality L. Our most practical scheme is achieved by setting $L = 1$, yielding read efficiency $O(N^{1/(s+1)})$ and space equal to $\Theta(N \cdot s)$. Note that for $L = N^{1/s}$ (which gives constant read efficiency), we can prove our scheme is secure using as leakage only the size of the access pattern (as used in previous works). *Assuming no keyword list has size more than $N^{1-1/\log\log N}$.***Assuming no keyword list has size more than $N^{1-1/o(\log\log\log N)}$.***For a keyword-list with size $n = N^{1-\epsilon(n)}$.

for example, OneChoiceAlloc, TwoChoiceAlloc and TwoChoiceAlloc+.

Our Practical Locality-Aware SE Scheme. In this chapter, we present our practical SE scheme for private keyword and database search (point queries) with tunable locality. For a parameter s that controls the space (the space is $s \cdot N$), our most efficient scheme has $O(1)$ locality and $O(N^{1/(s+1)})$ read efficiency. In particular, our scheme achieves up to 577× less false positives for all practical database sizes when compared to OneChoiceAlloc in a large database (approximately 1 TB). This translates into big improvements in practical performance in an external memory setting. We stress here that our scheme's leakage profile is slightly different (not necessarily worse) than OneChoiceAlloc and its worst-case asymptotic read efficiency is worse than OneChoiceAlloc. There are two reasons that explain why we are bet-

ter in practice despite worse asymptotics: One reason is the hidden constants in the polylogarithmic complexity of OneChoiceAlloc. Second, and most importantly, our asymptotic complexity is *worst-case* (meaning that there are some queries that we answer optimally or close to optimally) while OneChoiceAlloc's complexities are tight (meaning that there are no queries that are answered optimally). Our scheme is designed to work fast in in-memory as well, where the bottleneck is not the random memory accesses, but computation. In particular it achieves up to 12× speed-up compared to the state-of-the-art in-memory SE scheme by Cash et al. [1]. Our scheme can also be tuned to achieve locality L and improved read efficiency $O(N^{1/s}/L)$. This is quite important in a parallel setting with L processing units. Using this tuning, each processor can have optimal locality and $O(N^{1/s}/L)$ read efficiency. Interestingly enough, for the case where $L = N^{1/s}$ (that gives optimal read efficiency), our scheme has exactly the same leakage profile as previous works.

3.1 Scheme with Optimal Locality

Our core scheme is inspired by the scheme of Asharov et al. [30], but is different is many ways. Asharov et al. proposed a scheme with optimal locality, optimal read efficiency and $O(N \log N)$ space. This scheme roughly works as follows: It uses $\ell = \log N + 1$ arrays $A_0, A_1, \ldots, A_\ell$[3] of size N. Array A_i consists of $N/2^i$ chunks and stores all keyword lists of size 2^i at randomly-chosen chunks (note it is assumed here that all keyword list sizes are powers of two—we do not have this assumption). This ensures that the size of data read from each array A_i is always the same, which

[3]The actual scheme uses hash tables instead of arrays but we use arrays here for clarity.

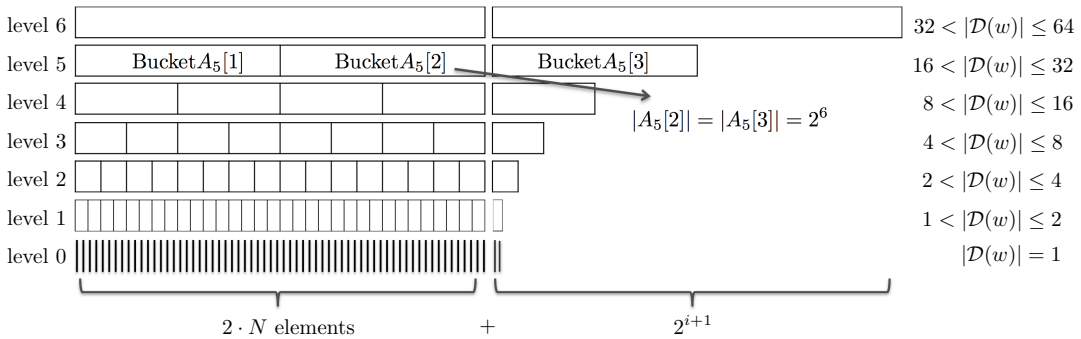


Figure 3.1: Example for $N = 64$ and $s = 7$. When $s = 2$, our scheme with optimal locality stores only levels 6 and 3, mapping all the queries of levels $0, 1, 2$ to level 3 and all the queries of levels $4, 5$ to level 6. The worst case read efficiency is $N^{1/2} = 8$ occurring when we map a query of size 1 to level 3.

is important for security. Therefore, to retrieve the results for a certain keyword w, one needs to read the right bucket at level i that contains the list. This bucket number is stored in an encrypted form in a separate dictionary and can be retrieved using the token for the keyword w. It is easy to show that such an approach reaches the aforementioned bounds.

Our main idea is to reduce the space of the above scheme by storing only s evenly distributed levels, where s is a small constant in practice (e.g., in our experiments we set $s = 2$ or 4). In particular we pick $p = \lceil \ell/s \rceil$ and we store only the levels $\mathcal{L} = \{\ell, \ell - p, \ell - 2p, \ldots, \ell - (s-1) \cdot p\}$. However, this creates many issues. For example, if level i is not stored, then the queries of size 2^i can no longer be answered. To avoid this problem we choose to store at level $i \in \mathcal{L}$ keyword lists $\mathcal{D}(w)$ such that

$$2^j < |\mathcal{D}(w)| \leq 2^i \,,$$

where $j \in \mathcal{L}$ is the smaller level following i in $\mathcal{L}$. (We stress that if i is the smallest level we ignore the lower bound in the above relation.) To store a keyword list whose

size falls in the above range, we pick a random bucket at level i that has enough space (we never split a keyword list across two buckets). While this looks like an easy fix, it creates further problems as we detail in the following paragraph.

In particular, we can no longer guarantee that *all* keyword lists with sizes $(2^j, 2^i]$ can fit in a single bucket at level i, which is important for maintaining our optimal locality. This is because, depending on the order that we store the keyword lists, there might be one that will have to reside in different buckets, ruining our optimal locality. For example, assume two consecutive levels that we store are levels 1 and 3 and the total number of elements we have is $N = 16$. There are three keywords in our data set w_1, w_2 and w_3, with $|\mathcal{D}(w_1)| = |\mathcal{D}(w_2)| = 4$, and $|\mathcal{D}(w_3)| = 8$. All these lists will be stored at level 3, which has two buckets of size 8. If we choose to store w_1 and w_2 in different buckets, then w_3 will have to be divided across the two buckets, increasing its locality from 1 to 2. This also affects the security of the scheme since there exist inputs that could trigger the aforementioned overflow and others that could not.

We address this problem by slightly increasing the space of each level $i \in \mathcal{L}$ from N to $2N + 2^{i+1}$—see Lemma 1. In particular, we are doubling the size of each bucket in each level and adding one more bucket per level. This allows us to guarantee that regardless of the order of the input keyword lists there will always be enough space to store an entire keyword list in one bucket.

Lemma 1 *Assume level i can store $2N + 2^{i+1}$ entries and let $\mathcal{W}$ be the set of keywords with list sizes $\leq 2^i$. Regardless of the order in which we store keywords*

$w \in \mathcal{W}$ at level i, there is always going to be enough space within a single bucket (of size 2^{i+1}) for all keywords $w \in \mathcal{W}$.

Proof 1 *Let level i be split in at most $\Lambda + 1$ buckets, so*

$$2N + 2^{i+1} = \Lambda \cdot 2^{i+1} + y\,, \tag{3.1}$$

where $0 \leq y < 2^{i+1}$ is the size of the last bucket. We prove our claim by contradiction. Suppose there exists a keyword $w \in \mathcal{W}$ whose list has size $\kappa \leq 2^i$ and for which there is not enough space in any bucket of level i. This means that all Λ buckets in level i have been filled with at least $2^{i+1} - \kappa + 1$ items and the last bucket has been filled with at least $y - \kappa + 1$ items. In that case, if we count the number of items that have been considered so far we have

$$\begin{aligned}
\#\ \textit{items considered} \ &\geq\ \Lambda \cdot (2^{i+1} - \kappa + 1) + y - \kappa + 1 \\
&= \frac{2N + 2^{i+1} - y}{2^{i+1}} \cdot (2^{i+1} - \kappa + 1) + y - \kappa + 1 \\
&\geq \frac{2N + 2^{i+1} - y}{2^{i+1}} \cdot (2^i + 1) + y - 2^i + 1 \ \textit{(since } \kappa \leq 2^i\textit{)} \\
&= N + \frac{N}{2^i} + 2 + y - \frac{y}{2} - \frac{y}{2^{i+1}} \\
&\geq N \ \textit{(since } i \geq 0\textit{)}\,.
\end{aligned}$$

Therefore we show that the total number of items considered so far is at least N, which is a contradiction. □

The arrangement of the mappings in our scheme is shown in Figure 4.1. In

particular, we present two cases, where in both $N = 64$: (1) We keep all the levels by setting s=7 and store all the keyword-lists of size $|\mathcal{D}(w)|$ in level $\lceil \log |\mathcal{D}(w)| \rceil$. (2) We set $s = 2$ and follow our algorithm which keeps only levels 3 and 6. In the latter case, all keyword-lists of size less than or equal to 8 are mapped to level 3 and the remaining ones to level 6.

Complexities. Clearly the above approach answers queries with optimal locality. Also, the read efficiency is $O(2^{\log N/s}) = O(N^{1/s})$. To see that, note that the maximum penalty in terms of false positives is paid by keywords lists $\mathcal{D}(w)$ with size $2^j + 1$ which are answered by the buckets of size 2^i (this is in case $i > \ell - (s-1) \cdot p$, i.e., i is not the last level). Therefore, by the definition of read efficiency

$$R \leq \frac{2^i}{2^j + 1} < 2^{i-j} \leq 2^{\lceil \lceil \log N \rceil / s \rceil} = O(2^{\log N/s}) = O(N^{1/s}) \,.$$

For $i = \ell - (s-1) \cdot p$ we have $R \leq 2^i$ but since $\ell - (s-1) \cdot p \leq p$ we have the same bound. The space of this approach is $O(s \cdot N)$.

3.2 Tuning the Locality of Our Scheme

Our scheme above achieves optimal locality. However, there are scenarios that we might want to increase slightly the locality to gain in read efficiency—this could be a parallel processing setting. One naive way to increase locality from 1 to L so that to gain in read efficiency is to partition our original data set into N/L data sets and apply our optimal locality scheme separately on each one of the smaller datasets. By using our previously described scheme, this would yield

$O((N/L)^{1/s})$ read efficiency (actually, this approach of data partitioning can work for every locality-optimal scheme). We propose here a new scheme with read efficiency $O(N^{1/s}/L)$ and locality L. This is much better in practice, and asymptotically better for any $L = \omega(1)$. The idea is as follows:

In our previous scheme, we chose to store at level i lists that have sizes in $(2^j, 2^i]$, where i and j are adjacent levels in $\mathcal{L}$ with $i > j$. To achieve locality L, we can choose to store at level i keyword lists $\mathcal{D}(w)$ such that

$$L \cdot 2^j < \mathcal{D}(w) \leq L \cdot 2^i \,.$$

(Again, if i is the smallest level we ignore the lower bound in the above relation.) Among those lists, the ones with size $\leq 2^i$ are stored as in the previous scheme; The ones with size $> 2^i$ are split into multiple chunks of size 2^i and one chunk of size less than 2^i. Then these chunks are stored as before. Note that because we again end up storing chunks of size $\leq 2^i$ at level i, Lemma 1 can be recast and still holds, guaranteeing that even with this new, modified algorithm, there will always be a bucket with enough space to store the relevant chunks.

Complexities. The locality for keyword lists of size greater than 2^i is at most L, while the read efficiency for those lists is optimal. For keyword lists of size less or equal to 2^i, the maximum penalty is achieved for the list of size $L \cdot 2^j + 1$, in which case the read efficiency is

$$R \leq \frac{2^i}{L \cdot 2^j + 1} < \frac{2^{i-j}}{L} = O\left(\frac{N^{1/s}}{L}\right) .$$

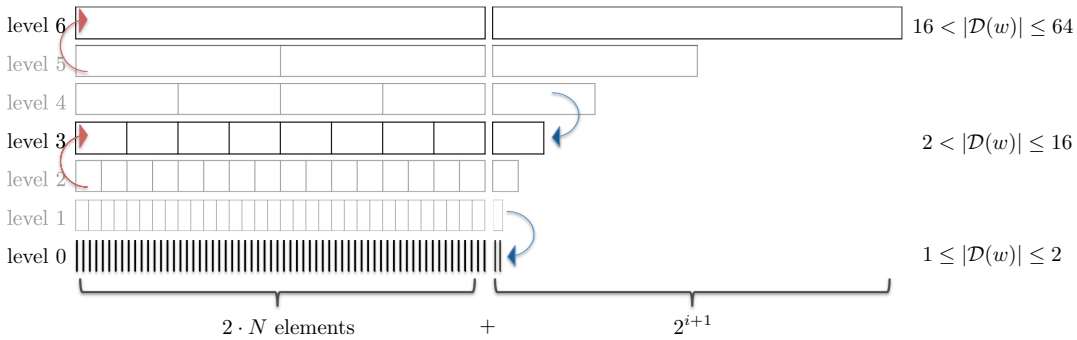


Figure 3.2: Example for $N = 64$, $s = 2$ and $L = 2$. Our scheme stores levels $0, 3, 6$. The red arrows depict the queries whose answers contain false positives but with optimal locality, while the blue arrows show queries with optimal read efficiency and constant locality.

Again, the above holds for $i > \ell-(s-1)\cdot p$. As opposed to before, for $i = \ell-(s-1)\cdot p$ the above, improved bound does not hold (in particular it is $O(N^{1/s})$), since keyword lists with size 1 must be answered by level i. To avoid that, we also keep level 0, which answers keyword lists with size $\leq L$. See Line 1 of Algorithm Setup. Only the size of the array A_0 can be N, instead of $2N + 1$.

The space remains $O(s \cdot N)$. Note our initial scheme with optimal locality is a special case of the above scheme for $L = 1$. The detailed algorithms of our schemes are shown in Figures 3.3 and 3.4. Figure 3.2 illustrates an example for $s = 2$ and $L = 2$. Note that algorithm Setup takes as input the parameters L (locality) and s (number of levels kept). We observe that our scheme keeps levels 0,3,6. The red arrows illustrate that these queries will be answered by the level above (including false positives), while the blue arrows introduce our new policy. In our new policy, given a stored level i, the $\log L$ levels above it will be stored and answered by the level i. For example, a keyword list of size 16 will be divided into two chunks of size 2^3 and each of these chunks will be stored in level 3.

Technical Details of Our Construction. In Figures 3.3 and 3.4 we illustrate our construction in more detail. In particular, the KeyGen algorithm takes as input the security parameter and computes the secret keys that are used in the randomized encryption scheme and the pseudorandom functions. The Setup algorithm takes as input the document collection, the secret keys, and the parameters s and L and in lines 1-6 it initializes the encrypted dictionary (it uses a hash table for the implementation of the encrypted dictionary) and the arrays A_i (each array A_i contains buckets of size 2^{i+1} — a collection of consecutive cells where each cell stores an encrypted (w, id) pair). In lines 7-15 the Setup algorithm places the keyword lists in the arrays A_i, while storing in the dictionary the bucket in which a keyword list or a chunk of the keyword list is stored. Finally, in lines 17-20 the entries of a bucket are randomly permuted and the arrays A_i are encrypted; each entry is encrypted using RND encryption and the produced key is a function of the keyword. This approach is only used in the keyword search scenario where we expect from the server to directly output the document identifiers. However, in the database search scenario, we encrypt each entry using RND encryption without choosing a key as a function of the keyword, since the server does not perform any decryption but outputs only the encrypted result to the client. In Figure 3.4, the Search algorithm executes the Token algorithm which produces the corresponding query tokens which are used in order to locate the entry of the dictionary which corresponds to the queried keyword. The same algorithm partially decrypts L entries of the dictionary to further detect the correct buckets and the correct array A_i and filters out the false positives; the server attempts to decrypt all the entries inside a bucket but

only the entries containing the queried keyword will have the last λ-bits to be 0. The same procedure could be used in the database search scenario to obtain the tuple identifiers of the answer. However, in the next section we provide a more efficient optimization to address this task since the tuples can be stored directly in the encrypted arrays A_i.

Scheme with Read Efficiency $O(R)$. It is easy to see that the above scheme can be tuned to achieve read efficiency $O(R)$ and worst-case locality $O(N^{1/s}/R)$ by setting $L = N^{1/s}/R$. We also note that for $L = N^{1/s}$ our scheme has optimal read efficiency $O(1)$ and has exactly the same leakage profile with prior SE schemes (such as Scheme 2). In the latter case, if we change the arrays A_i into hash tables, then the encrypted dictionary is not required.

3.3 Optimizations of Our Scheme

We now describe various optimizations that can be applied to our scheme. The first two are used in our implementation.

Optimization 1 - Decryption of the Result at the Client. In the current scheme, the decryption of the result (namely the identifiers of the documents containing the searched keyword) take place at the server side (see Line 6 of Algorithm Search). In particular the server performs 2^{i+1} decryption attempts, where 2^i is the size of the bucket from which we retrieve the answer. We can reduce the decryption cost to be proportional to the size of the result, by assigning the decryption to the client. This optimization can be used in keyword search by increasing the number of

$\underline{\mathbf{k} \leftarrow \mathsf{KeyGen}(1^\lambda)}$

1: $(k_1, k_2, k_3) \leftarrow^{\$} \{0,1\}^\lambda$.
2: $\mathsf{RND} = (Enc, Dec)$ is CPA-secure encryption scheme.
3: $F : \{0,1\}^* \times \{0,1\}^\lambda \to \{0,1\}^*$ is a PRF and $\mathsf{H} : \{0,1\}^* \to \{0,1\}^*$ is a collision resistant hash function.
4: **return** (k_1, k_2, k_3).

$\underline{\mathcal{I} \leftarrow \mathsf{Setup}(\mathbf{k}, \mathcal{D})}$

1: Parse $\mathbf{k}$ as (k_1, k_2, k_3). Let s and L be publicly-known parameters.
2: Let $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$ and $\ell = \lceil \log N \rceil$. Set $p = \lceil \ell / s \rceil$. Let $\mathcal{L} = \{\ell, \ell - p, \ldots, \ell - (s-1) \cdot p\}$. If $L > 1$ set $\mathcal{L} = \mathcal{L} \cup \{0\}$.
3: Initialize a hash table HT that can store up to N elements.
4: **for** each evenly distributed level $i \in \mathcal{L}$ **do**
5: Initialize an array A_i of size $2N + 2^{i+1}$.
6: Divide A_i in Λ_i buckets of size 2^{i+1} and one bucket of size $y_i < 2^{i+1}$.
7: Let $A_i[1], A_i[2], \ldots, A_i[\Lambda_i + 1]\}$ be the set of those buckets.
8: **for** each keyword $w \in \mathbf{W}$ in a random order **do**
9: Find adjacent j and i in $\mathcal{L}$ such that $L \cdot 2^j < |\mathcal{D}(w)| \leq L \cdot 2^i$ (if i is the smallest level, we ignore the lower bound).
10: Split $\mathcal{D}(w)$ into a set $\mathbf{C}_w$ of chunks containing q_w chunks of size 2^i and one chunk of size $r_w < 2^i$.
11: $count = 0$.
12: **for** each chunk $c \in \mathbf{C}_w$ **do**
13: $count = count + 1$.
14: Let $\mathbf{A}$ be the set of buckets in A_i that have enough space for chunk c (say those are $A_i[j_1], A_i[j_2], \ldots, A_i[j_f]$).
15: Pick one bucket $a \in \mathbf{A}$ (say $A_i[x]$) uniformly at random and store c in a at the first available position.
16: $HT.\mathbf{add}(\mathsf{H}(F_{k_1}(w)||count), [i||x] \oplus \mathsf{H}(F_{k_2}(w)||count))$.
17: Add random $(key, value)$ pairs to HT so that the total number of elements it stores is N.
18: **for** each stored array A_i (i.e., $i \in \mathcal{L}$) **do**
19: **for** each bucket $b \in A_i$ **do**
20: Randomly permute all entries (w, id) within b.
21: Replace each entry (w, id) of b with $\mathsf{RND}.Enc_{key}(id||0^\lambda)$ where $key = F_{k_3}(w)$.
22: Let $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$.
23: **return** $(HT, \mathcal{A})$.

Figure 3.3: KeyGen and Setup algorithms for our scheme with locality L, read efficiency $O(N^{1/s}/L)$ and space $\Theta(s \cdot N)$.

$t \leftarrow \mathsf{Token}(\mathbf{k}, w)$ ▷ Token is executed by the client.

1: Parse $\mathbf{k}$ as (k_1, k_2, k_3).
2: tag $\leftarrow F(k_1, w)$, vtag$\leftarrow F(k_2, w)$, etag$\leftarrow F(k_3, w)$.
3: **return** (tag,vtag,etag).

$(\mathcal{X}, \mathbf{k}, \mathcal{I}) \leftrightarrow \mathsf{Search}(\mathbf{k}, w, \mathcal{I})$

1: $t \leftarrow \mathsf{Token}(\mathbf{k}, w)$.
2: Parse t as (tag,vtag,etag) and $\mathcal{I}$ as $(HT, \mathcal{A})$.
3: **for** $count = 1$ to L **do**
4: evalue $\leftarrow HT.\mathbf{get}(\mathsf{H}(\text{tag}||count))$.
5: **if** evalue is not NULL **then**
6: $[i, \mathsf{offset}] \leftarrow$ evalue $\oplus\ \mathsf{H}(\text{vtag}||count)$.
7: **for** all entries e in bucket $A_i[\mathsf{offset}]$ **do**
8: **if** $\mathsf{RND}.Dec_{\text{etag}}(e)$ outputs $id||0^{\lambda}$ **then**
9: Add id to the result set $\mathcal{X}$.
10: **return** $(\mathcal{X}, \mathbf{k}, \mathcal{I})$.

Figure 3.4: Token and Search algorithms for our scheme with locality L, read efficiency $O(N^{1/s}/L)$ and space $\Theta(s \cdot N)$.

interactions; one round is required for the server to send the encrypted document-ids to the client and the client to decrypt them in order to request from the server in the second round to return the actual documents. In database search we only need one round of interaction, since using RND allows us to directly encrypt and store the tuples (instead of tuple-ids) in the encrypted index. It is highlighted that in this case the server does not perform any partial decryption of the tuples, i.e., the server identifies a super set of the result and returns this to the client (this super set of the result may contain false positives, which the client is responsible to filter out). This optimization requires the following changes:

- First, do not permute the entries in the bucket, as is done in Line 19 of Algorithm Setup. This will not violate our security since we will not allow any decryptions to take place at the server side.

- Second, instead of encrypting each entry in the bucket with a secret key derived from the respective keyword (see Line 20 of Algorithm Setup), encrypt all entries across all buckets with a single secret key that is stored at the client and is never revealed to the server.

- Finally, during the search, the server just sends the whole bucket encrypted to the client. Since the entries within the bucket are not randomly permuted, the entries corresponding to each keyword are consecutive. If the client knows the *start* and *end* point of each keyword, then he can decrypt only the part of the bucket that contains the actual result while skipping the remaining cells.

We note here that in order for the client to get the *start* and *end* values, our scheme can use an extra encrypted dictionary (like the one used to store offset and level i). To avoid increasing the space though, one could store only one dictionary with the *start* and *end* information, retrieve those and then derive (from *start* and *end*) the bucket offset and the level and request those from the server. The above observations can significantly accelerate the search procedure. Note that the above improvements are not applicable in Scheme 2 since in their case each bucket contains pieces of the result in arbitrary positions.

Optimization 2 - Storing One Level Less. Recall that in our scheme with constant locality we answer queries with size bigger than $2^{\ell-p+1}$ by retrieving one bucket from level ℓ (which both have size $2^{\ell+1}$). Since $2^{\ell+1} \geq N$, the aforementioned queries require reading information that is at least equal to the size of the dataset. We propose not storing level ℓ, but only the levels $\ell - p, \ell - 2p, \ldots, \ell - (s-1)p$,

reducing the number of stored levels from s to $s-1$. To accommodate the queries with size $> 2^{\ell-p+1}$, we just have the server return the whole level $\ell - p + 1$. This does not asymptotically increase the read efficiency for level ℓ. In summary, with this new approach, we can store space $O(s \cdot N)$ for a read efficiency of $O(N^{\frac{1}{s+1}})$, and constant read locality. This optimization applies to the locality L scheme as well.

Optimization 3 - More Efficient Level Selection by Increasing Leakage. The basic algorithm for constant locality stores s out of $\ell = \lceil \log N \rceil$ levels which are evenly distributed. This means that selecting a level is a function of N. For instance, let us assume that $N = 2^{32}$ and $s = 4$. In this case our basic algorithm preserves levels 32, 24, 16, and 8. Yet in the case where the maximum keyword list has size 4, then we observe that all queries are answered in level 8 and the remaining levels have no use. Thus, given statistical information about the stored dataset we can construct a better level selection algorithm. Note that an SE scheme using this optimization should increase $\mathcal{L}_{\mathsf{SETUP}}$ by this additional statistical information. An example of such information that can be used is the minimum and maximum word lists. For real datasets this optimization can radically improve the performance of our proposed schemes, but in the experimental evaluation we do not consider it to present fair comparisons to related work.

Optimization 4 - Fault Tolerance. Existing fault-tolerance file system architectures are using the notion of replication to address failures. A typical replication factor for those systems is at least 3, meaning that the initial dataset with size N will be expanded three times. For instance, the default replication factor of Apache Hadoop File System (HDFS) is set to be 3 [51]. We can change our scheme to

replicate all keyword/id pairs in all s levels (Lemma 1 will still hold). In this way, for $s = 3$, our scheme can get fault-tolerance for free (since we are storing $s \cdot 2N$ space), while other schemes would have to explicitly triple the space they are using.

3.4 Security Analysis and Leakage Profile

Our main construction for general values of L leaks the following information (when searching fo keyword w):

1. $\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D})$, as defined in Section 2.6. This is leaked by all previous schemes.

2. A deterministic function of the queried keyword $id(w)$ (search pattern). This is leaked by all previous schemes.

3. The bucket identifier $bucket(w)$ where w is read from. In particular, $bucket(w)$ contains information about the portion of the memory read to retrieve the result of a specific keyword (the level i and the offset of the bucket), which depends on the order in which the keywords were considered in the Setup algorithm. We believe this leakage is not meaningful since the order is decided at random. This information is not leaked by previous schemes.

We emphasize here also that our scheme, due to the first optimization in Section 3.3 and as opposed to all previous schemes, *does not explicitly leak* the exact size of the access pattern (it just leaks an upper bound on the size of the access pattern through the size of the bucket read). To summarize, we can now formally write our

leakage functions as

$$\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}) = N \text{ and } \mathcal{L}^{new}_{\mathsf{SEARCH}}(\mathcal{D}, w) = (id(w), buckets(w)).$$

Leakage Functions for the Case $L = N^{1/s}$. Unlike the general case, our scheme with $L = N^{1/s}$ has exactly the same leakage profile as previous schemes (e.g., Scheme 2): it just leaks $\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D})$ and $\mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w)$ (or $\mathcal{L}^{DB}_{\mathsf{SEARCH}}(\mathcal{D}, w)$ in the case of the database scenario), as mentioned in Section 2.6.

Theorem 1 *Given F is a pseudorandom function, H is a collision-resistant hash function and RND is a CPA-secure encryption scheme, the SE scheme of Figures 3.3 and 3.4 is $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}^{new}_{\mathsf{SEARCH}})$-secure according to Definition 1 and in the random oracle model.*

Additionally, for the case when the locality $L = N^{1/s}$, the scheme is $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{SEARCH}})$-secure (or $(\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}^{DB}_{\mathsf{SEARCH}})$-secure in the case of the database scenario) according to Definition 1 and in the random oracle model.

Proof 2 *For simplicity, we provide the proof of Theorem 1 for constant L using the first optimization described in section 3.3 and for $L = N^{1/s}$ using $\mathcal{L}^{DB}_{\mathsf{SEARCH}}$ leakage (database scenario).*

The simulator algorithms:

- $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$
- $\mathsf{SimToken}(\mathcal{L}^{new}_{\mathsf{SEARCH}}(\mathcal{D}, w))$
- $\mathsf{SimToken}(\mathcal{L}^{DB}_{\mathsf{SEARCH}}(\mathcal{D}, w))$

are shown in Figure 3.5 and Figure 3.6, respectively.

For the first part of the proof (which is the same for both cases of Theorem 1) we must show that no PPT algorithm Dist *can distinguish, with more than negligible probability, between the index* $\mathcal{I}_{real} = (HT_{real}, \mathcal{A}_{real})$ *output by* $\mathsf{Setup}(\boldsymbol{k}, \mathcal{D})$ *and the index* $\mathcal{I}_{ideal} = (HT_{ideal}, \mathcal{A}_{ideal})$ *output by* $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$. *This is because*

- *Both* $\mathcal{A}_{real}$ *and* $\mathcal{A}_{ideal}$ *are sets of* s *arrays*

$$A_\ell, A_{\ell-p}, A_{\ell-2p} \ldots A_{\ell-(s-1)\cdot p}\,,$$

  where in both cases A_i *has* $2N + 2^i$ *entries (*A_0 *is included for* $L > 1$*).* $\mathcal{A}_{real}$ *contains the encryption of values using a CPA-secure scheme, while* $\mathcal{A}_{ideal}$ *contains random values of the same format.*

- *Similarly,* Dist *cannot distinguish* HT_{real} *from* HT_{ideal}*, since both have* N *entries,* HT_{real} *encrypts entries by "xoring" with the output of a pseudorandom function and* HT_{ideal} *contains random values.*

For the second part of the proof and according to Definition 1, we need to prove that there does not exist a PPT algorithm Dist *that can distinguish between the outputs of* $\mathsf{Search}(\boldsymbol{k}, w, \mathcal{I})$ *and the outputs of* $\mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}^{new}_{\mathsf{SEARCH}}(\mathcal{D}, w))$ *and* $\mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}^{DB}_{\mathsf{SEARCH}}(\mathcal{D}, w))$. *This is because:*

- *Both the* Search *and* SimSearch *produce the same tokens and operations for the same repeated keywords.* SimSearch *uses the search pattern leakage* Prev *included in* $st_{\mathcal{S}}$.

Case for locality $L = 1$.

- *For a given keyword,* SimSearch *and* Search *access the same* $[i, \mathsf{offset}]$ *pair, included in the* $\mathcal{L}^{new}_{\mathsf{SEARCH}}(\mathcal{D}, w)$ *leakage and therefore the distributions are trivially indistinguishable. Note that for* SimSearch *to access a specific* $[i, \mathsf{offset}]$ *pair we program the random oracle* H *accordingly—see Line 9 in Figure 3.5.*

Case for locality $L = N^{1/s}$.

- *For a given keyword,* SimSearch *and* Search *do not necessarily access the same* $[i, \mathsf{offset}]$ *pairs. However, the* $[i, \mathsf{offset}]$ *pairs accessed by both algorithms are distributed* identically. *In particular the real* $[i, \mathsf{offset}]$ *pairs accessed by* Search *are distibuted uniformly at random (due to such placement by* Setup*), while* SimSearch *chooses uniformly at random* $[i, \mathsf{offset}]$ *pairs among the pairs that she did not choose before—see Line 11 in Figure 3.6. Again, for* SimSearch *to output a specific* $[i, \mathsf{offset}]$ *pair dictated by the simulator we program the random oracle* H *accordingly—see Line 12 in Figure 3.6.*

3.5 Dynamic SE (DSE)

In this section, we present an extension to our current work that allows updates. Allowing updates is a twofold challenge because it requires the following: a) enable efficient inserts and deletes; b) avoid leaking extra information while achieving forward/backward privacy.

$(\mathcal{I}, st_{\mathcal{S}}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$

Let $N \leftarrow \mathcal{L}_{\mathsf{SETUP}}(\mathcal{D})$, $\ell = \log N$, $p = \lceil \ell/s \rceil$.
Let $\mathcal{L} = \{\ell, \ell - p, \ldots, \ell - (s-1) \cdot p\}$. If $L > 1$ set $\mathcal{L} = \mathcal{L} \cup \{0\}$.
$\mathbf{k} \leftarrow \mathsf{KeyGen}(1^{\lambda})$.
Initialize a hash table HT of size N random entries and mark them as "unrevealed".
for each evenly distributed level $i \in \mathcal{L}$ **do**
    Initialize an array A_i of size $2N + 2^{i+i}$ with random elements.
    Divide A_i in Λ_i buckets of size 2^{i+1} and one bucket of size $y_i < 2^{i+1}$.
    Let $A_i[1], A_i[2], \ldots, A_i[\Lambda_i + 1]\}$ be the set of those buckets and mark all the buckets as "unrevealed".
Let Prev an empty hash table.
Set $\mathcal{I}$ to be the set $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$ and encrypted dictionary HT.
Set $st_{\mathcal{S}}$ to include $\mathcal{I}$, Prev and $\mathbf{k}$.
return $(\mathcal{I}, st_{\mathcal{S}})$.

$(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I}) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}^{new}_{\mathsf{SEARCH}}(\mathcal{D}, w), \mathcal{I})$

Parse $st_{\mathcal{S}}$ as $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$, HT, Prev and $\mathbf{k} = k_1, k_2$.
Let $(id(w), buckets(w)) \leftarrow \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w)$.
if $\mathsf{Prev}.\mathbf{get}(id(w)) \neq \mathsf{null}$ **then**
    return $(\mathsf{Prev}.\mathbf{get}(id(w)), st_{\mathcal{S}}, \mathcal{I})$. ▷ $st_{\mathcal{S}}$ contains the exchanged client-server messages.
else
    Set $\mathcal{X} \leftarrow \emptyset$, tag $= F(k_1, id(w))$, vtag $= F(k_2, id(w))$ and $count = 1$.
    for each $(i, \mathsf{offset}) \in buckets(w)$ **do**
        Pick uniformly at random an "unrevealed" entry $e = (key, value)$ from hash table HT.
        Program the random oracle H such that $\mathsf{H}(\text{tag}||count) = key$ and $\mathsf{H}(\text{vtag}||count) = value \oplus [i, \mathsf{offset}]$.
        Add e to the set $\mathcal{X}$ and mark e as "revealed" and $count = count + 1$.
    Use (tag, vtag) to simulate the messages exchanged between the client and the server and store them in $st_{\mathcal{S}}$.
    $\mathsf{Prev}.\mathbf{add}(id(w), \mathcal{X})$.
    Update $st_{\mathcal{S}}$ with new values of Prev, the choices that the random oracle made.
return $(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I})$.

Figure 3.5: Simulator algorithms $\mathsf{SimSetup}$ and $\mathsf{SimSearch}$ for scheme with $O(L)$ locality and $O(N^{1/s}/L)$ read efficiency.

Proposed Solution. Our DSE scheme is based on a solution proposed by Demertzis et al. [2, 3] for Range SE schemes. It is also used by commercial databases,

$(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I}) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w))$

Parse $st_{\mathcal{S}}$ as $\mathcal{A} = \{A_i : i \in \mathcal{L}\}$, HT, Prev and $\mathbf{k} = k_1, k_2$.
Let $(id(w), |\mathcal{D}(w)|) \leftarrow \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w)$.
if $\mathsf{Prev}.\mathbf{get}(id(w)) \neq \mathsf{null}$ **then**
    return $(\mathsf{Prev}.\mathbf{get}(id(w)), st_{\mathcal{S}}, \mathcal{I})$. ▷ $st_{\mathcal{S}}$ contains the exchanged client-server messages.
else
    Find table A_i for maximum i such that $2^i < |\mathcal{D}(w)|$ (if $|\mathcal{D}(w)|$=1 set $i = 0$).
    $q = \lceil |\mathcal{D}(w)|/2^i \rceil$.
    Set $\mathcal{X} \leftarrow \emptyset$, tag $= F(k_1, id(w))$, vtag $= F(k_2, id(w))$.
    for count $= 1$ to q **do**
        Pick uniformly at random an "unrevealed" entry $e = (key, value)$ from hash table HT.
        Pick uniformly at random an offset of an "unrevealed" bucket b at level i.
        Program the random oracle H such that $\mathsf{H}(\text{tag}||count) = key$ and $\mathsf{H}(\text{vtag}||count) = value \oplus [i, \mathsf{offset}]$.
        Add e to the set $\mathcal{X}$ and mark e and b as "revealed".
    Use (tag, vtag) to simulate the messages exchanged between the client and the server and store them in $st_{\mathcal{S}}$.
    $\mathsf{Prev}.\mathbf{add}(id(w), \mathcal{X})$.
    Update $st_{\mathcal{S}}$ with new values of Prev and the choices that the random oracle made.
return $(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I})$.

Figure 3.6: Simulator $\mathsf{SimSearch}$ for scheme with $O(N^{1/s})$ locality and $O(1)$ read efficiency.

such as Vertica [52] (organizes the updates in log-merge trees). This commonly used technique is preferable for the following reasons: i) it can use our very efficient static SE scheme as a "black box", ii) it enables easy leakage formulations, iii) it captures forward/backward privacy. The leakage of this approach is essentially the entire history of the $\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{SEARCH}}$ leakages of every index that was once "active" at the server. The main idea is that we organize n sequential updates to a collection of at most $\log n$ independent encrypted indexes. In particular, for each new

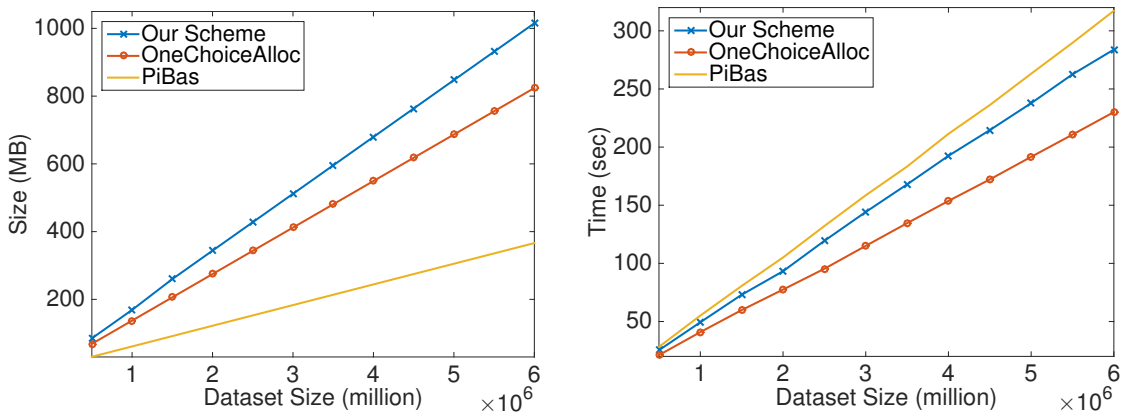
tuple, the data owner initializes a new SE scheme by creating a new SE index that contains only the specific tuple. The single-tuple index is subsequently uploaded to the untrusted server. Whenever two indexes of the same size t are detected there are downloaded by the data owner, decrypted and merged to form a new SE index of size $2t$, again with a fresh secret key. The new index is then used to replace the two indexes of size t. Clearly, a merge may have a cascading effect, i.e. subsequent merges. In this case, all merges are executed at the same time to avoid redundant work, that is constructing and uploading intermediate indexes. Deletions are simulated by inserting cancellation tuples. For further details, we refer the reader to [2, 3]. The space is linear in the size of the input and the number of updates. The locality of this approach is $O(L \cdot \log n)$ and the read efficiency is $O(N^{1/s}/L)$. In Chapter 5, we discuss in more depth how we can transform a static SE to a DSE with forward and backward privacy.

3.6 Experiments

In this section we experimentally evaluate the performance of our proposed scheme. We compare our scheme only with linear-space approaches. If one can afford $N \log N$ space, the best scheme (both asymptotically and in practice) is Scheme 1 of Asharov et al. [30]. As such, we compare our work with the static construction of Cash et al. [1] and Scheme 2 of Asharov et al. [30]—see Table 3.1. We refer to the former as *PiBas* and to the latter as OneChoiceAlloc. We compare our scheme only with the basic construction of Cash et al. [1] (i.e. *PiBas*) because the more optimized

proposed schemes (with good locality) are sub-optimal compared with the schemes of Asharov and introduce new leakages (in $\mathcal{L}_{\mathsf{SETUP}}$ leakage). Nevertheless, our scheme can be tuned with the use of the third optimization presented in Section 3.3 to achieve the same performance as the most optimized scheme of Cash et al. [1]. Moreover, we do not compare our scheme with Scheme 3 of Asharov et al. [30] because it assumes that no keyword list has size more than $N^{1-1/\log\log N}$ as shown is Table 3.1. Instead, we explain in Section 3.6.4 why this assumption is not realistic and we experimentally show the superiority of our scheme compared to Scheme 3 of Asharov et al. [30], by adopting the same assumption.

We organize the experimental section as follows. Section 4.3.1 presents the experimental setting and the technical details of our implementation. Section 3.6.2 focuses on the comparison of our work with *PiBas* and OneChoiceAlloc in an in-memory setting, while Section 3.6.3 compares our scheme with *PiBas* and OneChoiceAlloc in an external memory setting where optimal read efficiency is guaranteed in our scheme and *PiBas* and optimal locality in OneChoiceAlloc. Then, we compare our scheme with OneChoiceAlloc, where optimal locality is guaranteed in both schemes, and we focus on the comparison of the number of false positives in both approaches. Finally, in section 3.6.3 we provide experiments in parallel scenarios where we can tune locality to further reduce the number of false positives while assuring optimal locality per parallel processing unit (in particular, for overall locality L, we can have L parallel processing units with constant locality per unit).



(a) Index size (b) Construction time

Figure 3.7: Index costs

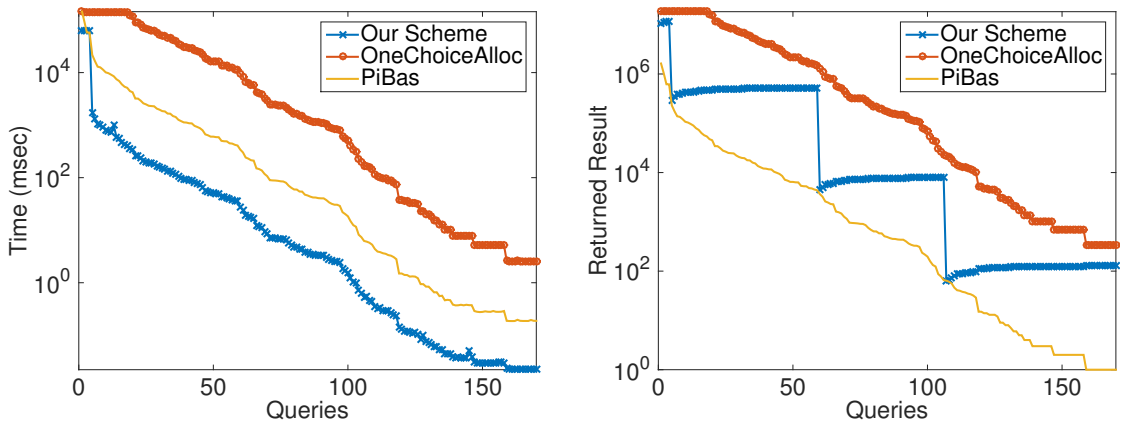
3.6.1 Setup

We carried out the implementation of our scheme, as well as the implementation of *PiBas* and OneChoiceAlloc in Java and conducted our experiments on a 64-bit machine with an Intel Xeon E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto library and the bouncy castle library [53] for the cryptographic operations. In particular, for the PRF and randomized symmetric encryption implementations we used HMAC-SHA-256 and AES128-CBC respectively for encryption. For our in-memory experiments we used single-threaded implementations, since a parallel implementation would favor our scheme compared to the schemes of our competitors. The experiments were conducted on a real dataset [46] consisting of $6,123,276$ records of reported incidents of crime. We consider the query attribute to be the *location description* which contains 173 distinct keywords (this is the x-axis in Figures 3.8(a),3.9). Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency comprises $1,631,721$

records. The specific dataset is used for the in-memory setting comparison. Our external memory experiments use the above dataset for the comparison with optimal read efficiency and locality $O(N^{1/s})$. For the external memory comparison with optimal locality we created a synthetic dataset. Note that in both, our locality-optimal scheme and OneChoiceAlloc the only factor that affects the number of false positives is the number of records. Thus, we create two synthetic datasets where the first contains $N = 2^{37} - 1$ records and one keyword list for each possible size which is a power of 2, such that $|\mathcal{D}(w_i| = 2^i$. The second synthetic dataset has the same data structure, only now it comprises $N = 2^{47} - 1$ records.

Implementation Details. We implement our locality-optimal algorithm using the first two optimizations described in Section 3.3. In particular, the client sends a token to the server and the server uses this token to locate and return the chunk that contains the answer of the query. This means that it is the responsibility of the client to decrypt and filter out the resulting false positives. We use optimization 1 to reduce the client's workload. In addition, we use optimization 2 to further reduce the required server space. We implement our read efficiency optimal algorithm without encrypted dictionary using for A_i hash tables.

The implementation of *PiBas* is straight-forward and was carried out as proposed in the work of Cash et al. [1]. We implemented the work of Asharov using a dictionary as was proposed for the general case. In particular, the dictionary contains the size of the result for each keyword. Thus, the client sends a token to the server and the server using the dictionary locates the first bucket in which the first result is placed and also the number of total buckets that it has to return to the



(a) Search Time (b) False Positives

Figure 3.8: Search costs

client. The client is responsible for decrypting the buckets in order to filter out the false positives.

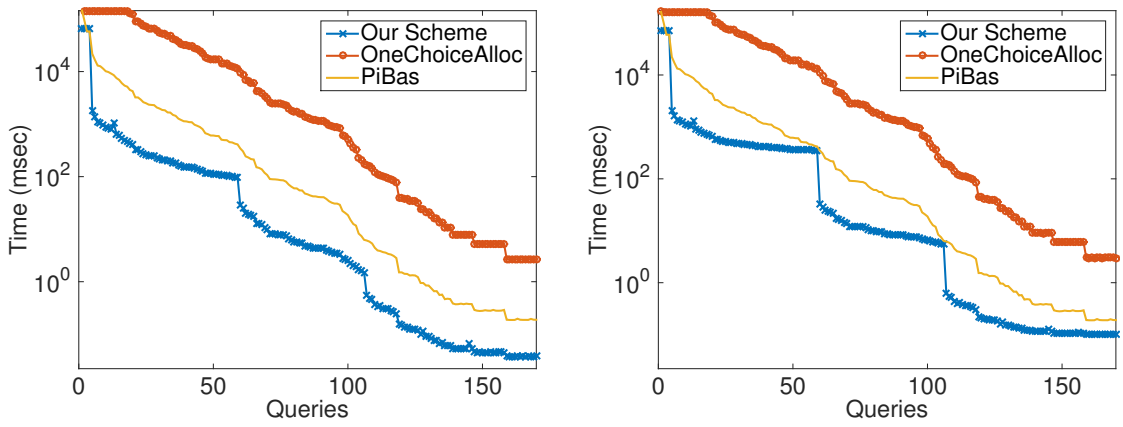
Both our scheme and OneChoiceAlloc can be implemented to return the exact result without any false positives as shown in Figures 3.3 and 3.4, where the server herself can filter out the false positives. However, doing this decreases the performance of both schemes. Only for the experiment in Figure 3.10(b), we implement OneChoiceAlloc to filter out the false positives at the server, since our read efficiency optimal scheme and *PiBas* do not contain false positives (we also provide an experiment for OneChoiceAlloc where the client performs the filtering — see Figure 3.10(a)). Note that carrying out the filtering on the server can be an efficient solution when the application has strict bandwidth limitations, because then the false positives are removed before transferring the data over the network.

3.6.2 In-memory Comparison

Index Costs. In the first set of experiments we evaluate the required index size and construction time of our scheme for different dataset sizes N. The results are shown in Figures 4.5(b) and 4.5(a) respectively. The construction time includes the I/O cost of transferring the dataset from the disk to the main memory, while the index size represents only the size of the encrypted index, since the size of the encrypted documents (or tuples) is the same for all schemes. Moreover, we partition the initial dataset into 12 sets of 500K tuples each, chosen uniformly at random from the entire dataset. Then, we begin with the first partition and consider the other partitions in each step in order to represent the construction time and index size as the size of the input gradually increases. For the initialization of our scheme we selected $s = 2$. According to our analysis and using the first optimization implies storing 2 levels and having read efficiency $O(N^{1/3})$. In this way we have space requirements comparable to OneChoiceAlloc whose space is approximately $3N$, while our case requires approximately $4N$ space; in both cases an encrypted dictionary of size N is required. Recall that due to Lemma 1, our scheme requires to store in each preserved level i an array of size $2N$ and one extra chunk of size 2^{i+1}. These space requirements are included in our figures. As expected, our schemes require slightly more storage and time for constructing the index compared to OneChoiceAlloc. We observe that *PiBas* requires less storage than both our scheme and OneChoiceAlloc, but the construction time performance is worse because of the need for more PRF evaluation per keyword/identifier pair. In particular, our schemes require index

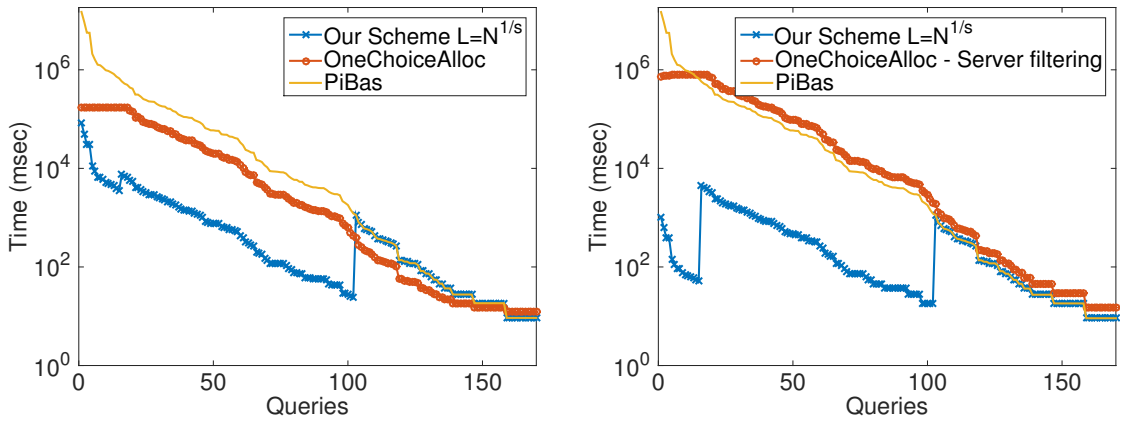
size from 85 to 1015 MB and construction time from 25 to 283 sec, while *PiBas* requires index size from 31 to 366 MB and construction time from 28 to 317 sec and OneChoiceAlloc requires index size from 69 to 824 MB and construction time from 21 to 230 sec. In addition to these outcomes though, it is worth mentioning that in our scheme the Setup algorithm is highly parallelizable, and therefore the cost of the specific algorithm can be distributed to different machines. In the case of *PiBas* and OneChoiceAlloc it is not straight-forward how we could parallelize the construction of the index.

Search Cost. Figure 3.8(a) illustrates the total time required by the server and client to perform every possible query excluding the communication cost. All schemes require transferring the result size through the network. However, our scheme and OneChoiceAlloc need to transfer more data than *PiBas* for each query. Our approach transfers on average 35× more information compared with *PiBas* and in the worst case this number becomes equal to 126×, while OneChoiceAlloc always transfers 324× times more data compared with *PiBas*. More specifically, Figure 3.8(b) shows the number of records returned by our approach, OneChoiceAlloc approach and *PiBas* approach, which transfers the exact result without false positives. For visualization purposes, we sort the queries based on their result size and we query each of them. Our schemes reached up to 12× speed-up compared to *PiBas* and achieved 347× speed-up in comparison with OneChoiceAlloc. This experiment confirms our non-trivial claim that optimal locality can be successfully used to achieve more efficient SE schemes even for in-memory architectures or fast external storage devices, i.e. solid state drives. Note that our scheme yields the



(a) 1 Gbps transfer rate (b) 200 Mbps transfer rate

Figure 3.9: Search Time



(a) End-to-end search time (b) Server search time

Figure 3.10: External memory comparison for the real dataset

above speed-up by reducing the workload of the server and client in the two following manners: i) the server is only responsible for returning the required chunks without evaluating a PRF for each result item; ii) the chunks contain the results of each query together, thus allowing her to decrypt only the requested result. These two features offered by our construction can, neither be integrated with *PiBas*, nor with OneChoiceAlloc.

The purpose of this experiment is to illustrate the amount of less work per-

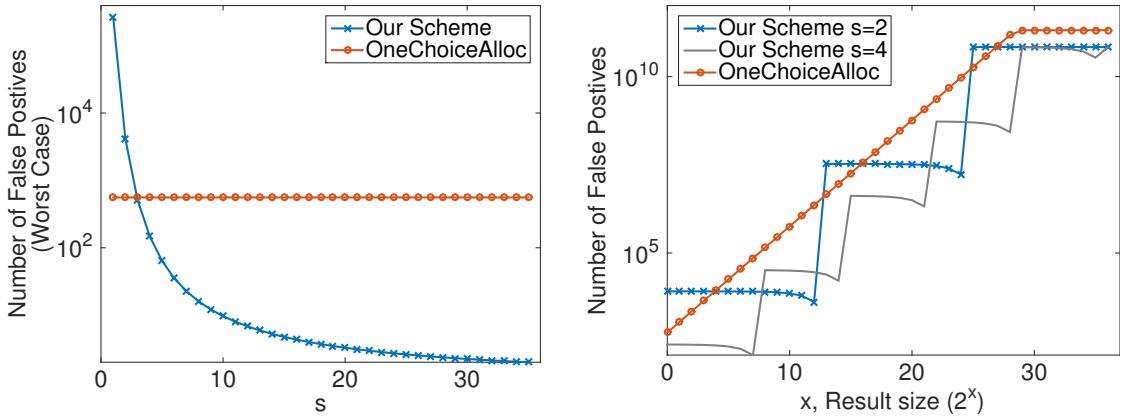
formed by the client and the server in our approach, and for this reason we exclude the communication cost. Nevertheless, we also conducted the experiments while taking into account the communication cost and observed that having on average 1 Gbps transfer rate in Figure 3.9(a) yielded results similar with the ones presented in Figure 3.8(a) when we compared our scheme with *PiBas* and OneChoiceAlloc. Additionally, for 200 Mbps transfer rate or more our scheme becomes the most efficient one for all possible queries as shown in Figure 3.9(b), while when the transfer rate is less than 200 Mbps some queries may become slower than *PiBas*. In comparison with the OneChoiceAlloc scheme our scheme is always more efficient regardless of the transfer rate, because our scheme transfers less false positives. For applications with limited communication bandwidth we suggest using the original protocol, where the server filters out the false positives herself , or the scheme with locality $L = N^{1/s}$. The latter scheme described in section 3.2 can be tuned to always be more efficient than *PiBas*. By tuning read efficiency to be optimal, less PRF evaluations are performed in our scheme compared with *PiBas*. Additionally, in the next section, we will show that this scheme is always more efficient than OneChoiceAlloc especially for queries with results size $> 2^7$ tuples.

3.6.3 External Memory Comparison

We have already mentioned that any scheme lacking locality cannot be used for big data applications (due to the cost of accessing data on the disk at random locations). In this section, we first compare our optimal read efficiency scheme (with

no-optimal locality $L = N^{1/s}$), with *PiBas* which has worst-case locality and OneChoiceAlloc which has optimal locality in external memory scenarios, where random accesses become the dominant factor. This comparison provides a very interesting outcome. Despite the fact that *PiBas* has worst-case locality and OneChoiceAlloc has optimal locality, both have similar performance, while our scheme achieves up to 60x faster search time compared to OneChoiceAlloc. Our scheme requires $5N$ space (N for level 0 and $4N$ for 2 additional levels — an encrypted dictionary is not required), while OneChoiceAlloc requires $4N$ space (including the encrypted dictionary). Then, we compare our optimal locality scheme with OneChoiceAlloc. In Section 3.6.2, we experimentally showed the superiority of our scheme over OneChoiceAlloc. We now measure only the number of false positives produced by each approach, since it is the only factor that differs between the two schemes and impacts their performance in practice.

Figure 3.10(a) depicts the end-to-end search time for the real dataset. *PiBas* and our scheme return to the client the exact answer, while in this experiment OneChoiceAlloc returns the answer with false positives. We observe that OneChoiceAlloc is faster than *PiBas*, while our scheme is up to 60x faster than OneChoiceAlloc and for the queries with size $\leq 2^7$ tuples it has the same performance as *PiBas* and at most 2.8x speed-down compared to OneChoiceAlloc. This is due to the block size whose size is 4K and can contain at most 2^7 encrypted keyword,id pairs. If the result size is smaller than 2^7 we still have to retrieve a whole block from the disk. Furthermore, if level 7 is the next stored level after 0 , then a query with size 2^6 requires from the server to read 2^6 blocks and for each of these blocks to perform a



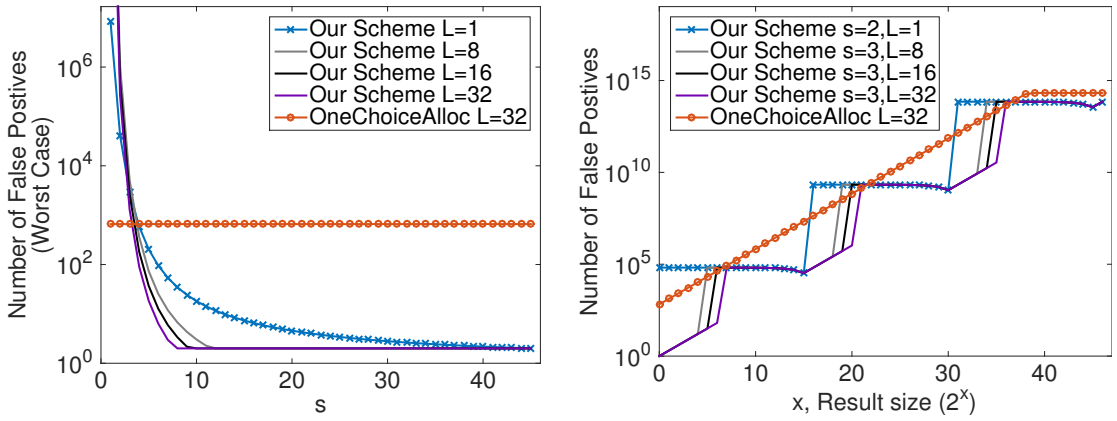
(a) FP for variable s

(b) FP for different sizes

Figure 3.11: External memory comparison ($N = 2^{37} - 1$)

random access. In this case, for all query results with less than 2^7 tuples our scheme works identically to the scheme of *PiBas*.

In this experiment, we implemented OneChoiceAlloc to conduct the filtering of the false positives on the server side (as was originally proposed in the paper [30]). Now, *PiBas*,OneChoiceAlloc and our scheme return to the client the result without false positives, so the communication cost and client work become the same for all schemes. Figure 3.10(b) compares the server search time for the threes schemes. We surprisingly observe that when the server filters the false positives, then for the majority of the queries *PiBas* is slightly better than OneChoiceAlloc. OneChoiceAlloc is designed to have optimal locality, while *PiBas* has worst-case locality. The reason, why in Figure 3.10(a) and in Figure 3.10(b) OneChoiceAlloc and *PiBas* have similar performance is because for each piece of the result *PiBas* pays a PRF evaluation (approximately 43 μsec) and a random access (approximately 10 msec), while OneChoiceAlloc requires $3 \log N \log \log N$ PRF evaluations (approximately 13msec per result item) and no random accesses. OneChoiceAlloc has optimal locality, but



(a) FP for variable s

(b) FP for different sizes

Figure 3.12: External memory comparison using parallelism ($N = 2^{47} - 1$)

requires significantly more cryptographic operations and the benefit gained from locality is nullified by the cost of the additional cryptographic operations. In Figure 3.10(b), we observe that our scheme requires up to 4 orders of magnitude less search time on the server than OneChoiceAlloc.

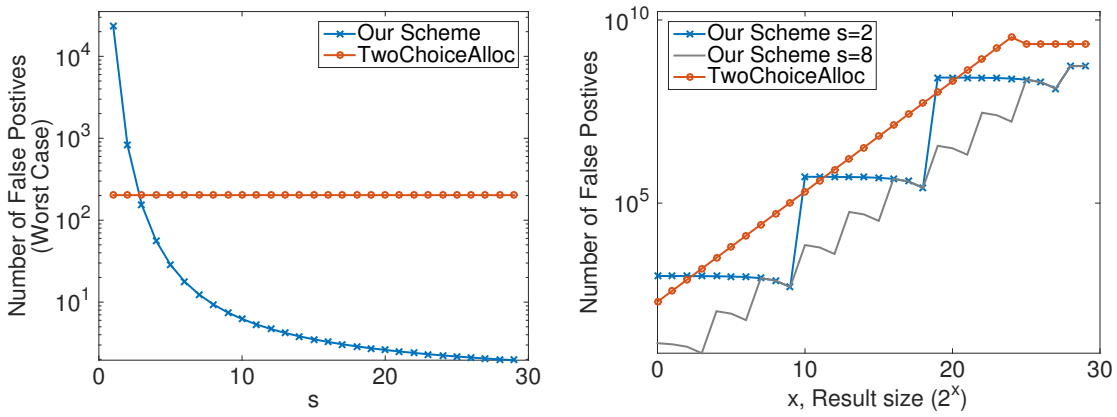
Below, our scheme has again optimal locality. Figure 3.11(a) depicts the worst-case read efficiency compared with OneChoiceAlloc for the first synthetic dataset of size $N = 2^{37} - 1$ and for different numbers of levels s in our scheme. For $s \geq 4$ our scheme always outperforms OneChoiceAlloc, and therefore we consider the interesting cases to be $s = 2$ and $s = 4$.

In Figure 3.11(b) we compare the number of false positives of our approach and OneChoiceAlloc, for all possible queries for $s = 2, 4$. For $s = 2$ (which requires the same space with OneChoiceAlloc), our approach outperforms OneChoiceAlloc for almost all possible queries, reaching maximum speed-up approximately 577×. This is because our scheme does not penalize queries with the worst-case bound.

We conduct the same experiments on a dataset of size close to 1 petabyte and

the resulting outcomes are illustrated in Figures 3.12(a)and 3.12(b) (see Our Scheme for $L = 1$ and the parallel OneChoiceAlloc for $L = 32$). Figure 3.12(b) shows that only for a small portion of all possible queries OneChoiceAlloc has less false positives than our scheme.In the worst case, our scheme reaches 86× speed-down compared to OneChoiceAlloc, but for the biggest portion we achieve significantly higher speed-ups up to 760×.

Since it is impractical to handle a dataset of size close to 1 petabyte without exploiting parallelism, we tune the locality L of our scheme to be equal to the number of parallel processing units (the locality per processor remains $O(1)$). In this case we always achieve a smaller number of false positives for all queries. For comparison reasons we created a parallel implementation of the OneChoiceAlloc scheme based on the (naive) idea proposed in Section 3.2. We could not further improve the parallel approach of OneChoiceAlloc. Figures 3.12(a) and 3.12(b) report the results of the same experiments, but now using 1, 8, 16, 32 parallel processing units for our scheme, and always 32 parallel processing units for the OneChoiceAlloc scheme. Recall that when $L > 1$ we also have level 0, but only for level 0 we store an array of size N instead of $2N + 2$. The vast improvement is achieved because for $s = 3$ our complexity is $O(N^{1/3}/L)$ (as explained in Section 3.2) while the complexity of OneChoiceAlloc is $\Theta(\log(N/L) \log\log(N/L))$. Thus, the impact of L in our scheme is much larger.



(a) FP for variable s

(b) FP for different sizes

Figure 3.13: External memory comparison ($N = 2^{47} - 1$)

3.6.4 Comparison with TwoChoiceAlloc

In the experimental evaluation section 4.3 we purposely did not include the comparison of our work with the Scheme 3 proposed by Asharov et al. in [30] (TwoChoiceAlloc), due to their assumption of not considering word lists of size more than $N^{1-1/\log\log N}$. This assumption cannot be taken into consideration for real world datasets. For instance, let us examine the real dataset of crime records that was used in our experiments that contains 21 attributes. Then, for 12 attributes out of the total of 21, their assumption is violated. Therefore, it becomes infeasible to use the TwoChoiceAlloc scheme on these attributes. Moreover, even though the assumption holds for the 9 remaining attributes note that these contain only unique values or have very small cardinality, i.e. the following attributes: id, case number, date and time, X coordinate, Y coordinate, longitude and latitude. For these attributes TwoChoiceAlloc can be used, but even then our scheme yield a smaller number of false positives. More specifically, we applied to our scheme the assumption of

$N^{1-1/\log\log N}$ by computing the level that has chunks bigger than the answer to a query of size $N^{1-1/\log\log N}$. Since it is impossible to consider such sizes (or bigger), we evenly select the appropriate levels out of the remaining excluding those that have chunks with size bigger than the answer of the query. This can be illustrated in an example where $N = 2^{47} - 1$. Then, the assumption that we do not have a query result of size bigger than $N^{1-1/\log\log N}$ means that the possible query sizes range from 1 to 2^{39}. Therefore, our algorithm will select evenly distributed levels starting from level 1 up to level 36 excluding all levels higher than 36. Figures 3.13(a) and 3.13(b) compare the two schemes showing that for the same amount of space which is equivalent to setting our redundancy factor $s = 2$ since TwoChoiceAlloc requires $4 \cdot N$ space, our scheme is always better than TwoChoiceAlloc. In addition, TwoChoiceAlloc assumes that the size of each word list is a power of 2 and N is also assumed to be a power of 2. Hence, in the worst case the TwoChoiceAlloc scheme requires padding the dataset, thus yielding a final size that reaches $8 * N$. In this case and in order for both schemes to use the same space we tune our redundancy factor to be equal to 8 so we also show the case of our scheme having redundancy factor $s = 8$.

Chapter 4: Efficient Searchable Encryption Through Compression

Since the first work on SE schemes proposed in 2000 [16], all follow-up works with linear size encrypted index (e.g., [1, 23, 26, 30]) require the server to perform cryptographic operations (PRF evaluations) to retrieve the result, the number of which is at least equal to the size of the query result. The main reason is that for security purposes a query result r is stored in $|r|$ random positions indexed by $|r|$ values each of them produced by a PRF. The server, given a token $t(w)$, has to expand it on $|r|$ sub-tokens (using PRF evaluations) to locate, retrieve and return to the client all the $|r|$ pieces of the result. A PRF evaluation corresponds to the most expensive operation in the search algorithm. The main question we are therefore asking in this Chapter is:

Can we design SE schemes that retrieve the keyword search result r with less than $|r|$ cryptographic operations?

It is worth noting that previous SE schemes with constant locality (ones that require few random accesses to retrieve the result) (e.g.,[30, 31, 32]) have partially addressed this problem by reducing the number of cryptographic operations required by the server and not the total number of cryptographic operations.

In this work we take a more aggressive approach and aim at reducing the *total*

number of cryptographic operations required by the protocol. We propose a novel SE scheme for private keyword and database search using compression that addresses the above question. Our SE scheme is the first in which the document identifiers matching a queried keyword can be retrieved with potentially less cryptographic operations than the result size $|r|$[1]. Informally we achieve that by storing an encryption of a compressed index instead of a traditional encrypted index. While combining compression and encryption can create security problems [54], we leverage the already existing leakage of searchable encryption to overcome this. We formally prove that our scheme can use any secure SE scheme as black-box (including the recent locality-aware SE schemes [30, 31, 32]), and any set of lossless compression algorithms improving the search efficiency of the used black-box by orders of magnitude without affecting its security.

We experimentally evaluate our scheme and show that, for the case of keyword search, it achieves up to **188**× speedup in terms of search time, compared to the most practical in-memory SE scheme. For the case of database search (where there are no overlaps across results and thus less structural leakage—see Section 2.5), we show that our saving is still high, up to **62**× for the *location description* attribute; up to **203**× for binary attributes (see section 4.3). Our SE scheme can be used as black-box in [2, 3] for further improving the performance of private range/aggregation queries and in [33] for improving boolean queries.

We combine our scheme with Oblivious RAM (ORAM) approaches and pro-

[1]Our scheme offers better search time for result sizes greater than 1; otherwise our scheme requires one cryptographic operation, just like other SE schemes. Thus, our scheme does not have performance benefits when all documents contain different words as well as in the database search for unique-key attributes.

pose Oblivious SE (OSE), a scheme that answers private keyword and point queries with ORAM-style security guarantees. Our OSE scheme reduces the index search time to access one million tuples using a state-of-the-art ORAM scheme approximately from 21 hours to 20 minutes. Our OSE scheme can be used in the recent works for oblivious querying processing [6, 55, 56] in order to further improve their performance.

4.1 Supported Leakage Functions

In total we present four SE constructions in this chapter that satisfy Definition 1 presented before. Each such scheme has different leakage, as we detail in the following.

- A simple SE construction for keyword search (SE-K);
- A simple SE construction for database search (SE-D);
- An ORAM-based SE construction for keyword search (OSE-K);
- An ORAM-based SE construction for database search (OSE-D).

Every construction leaks different amount of information. In Table 4.1 we show all the leakages in detail. We now explain the intuition behind these leakages.

Leakages for SE-K. Our simple SE construction for keyword search leaks only the size of the index N and number of the indexed documents n during setup. During a query for w, it leaks $(id(w), \mathcal{D}(w))$ where $id(w)$ is a random-looking λ-bit number that we map to each keyword w, called alias of w (capturing the *search pattern*,

construction	$\mathcal{L}_{\mathsf{SETUP}}$	$\mathcal{L}_{\mathsf{SEARCH}}$
SE-K	(N, n)	$(id(w), \mathcal{D}(w))$
SE-D	N	$(id(w), \lvert\mathcal{D}(w)\rvert)$
OSE-K	(N, n)	$\lvert\mathcal{D}(w)\rvert$
OSE-D	N	$\lvert\mathcal{D}(w)\rvert$

Table 4.1: Different leakages in our constructions.

i.e., whether a keyword query has been repeated or not). The set $\mathcal{D}(w)$ captures the *access pattern*, revealing which document overlaps between previously queried documents.

Leakages for SE-D. The main difference here is in the query leakage, where we *leak* the size of the access pattern, instead of the access pattern itself. Also, since $N = n$ in the database search scenario, only N is naturally leaked.

Leakages for OSE-K and OSE-D. As opposed to construction SE-K and SE-D, our ORAM-based constructions *only leak the size of the result that is returned*. We can consider this leakage to be *ideal* for any scheme that supports sublinear-time search, since in order to hide the size of the result we will need to either download the entire encrypted index, or equivalently to pad the result to the maximum size. In both cases the size of each query answer will be proportional to the input size, i.e. $O(N)$.

4.2 Our Approach

Our approach consists of two main steps: Given the dataset $\mathcal{D}$, in the first step we compress each list $\mathcal{D}(w)$; The second step uses the output compressed lists (for all keywords w) as input to the SE setup algorithm. When an encrypted search

query is performed, the accessed list is much smaller (due to compression)—once we receive the compressed list, we can decompress it and retrieve the result. Note that while asymptotically the time required for the search is the same as other schemes, the number of cryptographic operations are only proportional to the size of the compressed list—this leads to significant savings in practice as we show in the experiments. We now describe various components of our approach in more detail.

Uniform document identifier reassignments. Prior SE schemes assume that for each document identifier we pick a random string of τ bits. Note that performing compression on random strings leads to almost negligible compression. In our approach, instead of assigning a random string of τ bits to each document identifier we assign an id chosen uniformly at random from the range $[0, n-1]$, where n is the total number of documents in our collection. This uniform reassignment of document identifiers is equivalent to having random strings of τ bits (this will be part of our security proofs). However, our approach leads to more efficient compression of the encrypted keyword lists.

Compression and partitioning. As mentioned earlier, instead of storing $\mathcal{D}(w) = \{id_1, \ldots, id_s\}$ in the encrypted index, we will store an encrypted version of that list, namely the string $\mathcal{Y} = \mathsf{Compress}(\mathcal{D}(w))$, computed using some compression algorithm. Note that $\mathcal{Y}$'s size is at most $s \log n$ bits meaning that the number of cryptographic operations required to retrieve the compressed result is reduced—the actual result can be easily retrieved from the compressed result with *no* cryptographic operations.

We further partition $\mathcal{Y}$ to chunks of $\lambda - \log n$ bits (we use $\log n$ bits to store

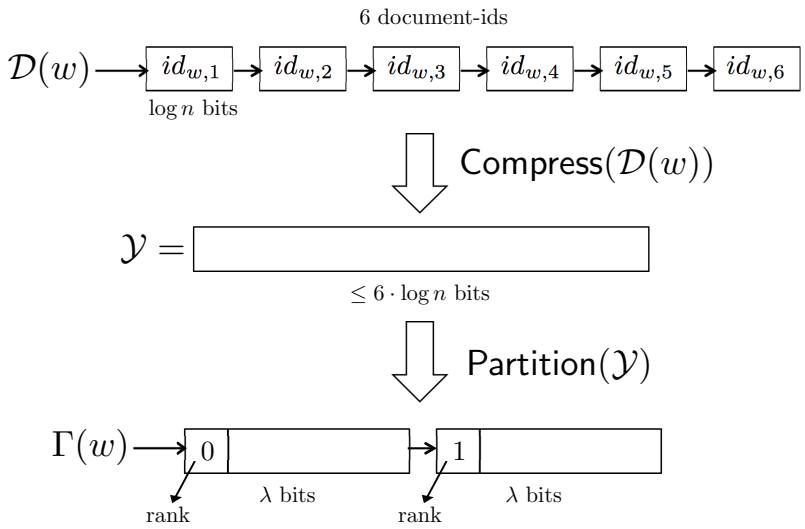


Figure 4.1: Our scheme first compresses the keyword lists and then performs the partitioning. Note that the packed words need to be stored with a rank, so that the decompression can work correctly.

the rank of the chunk as we explain below), where λ is the security parameter. This assures we "pack" the maximum amount of information into one PRF evaluation. For example, we can partition $\mathcal{Y}$ into words G_1, G_2 etc., ending up with approximately $\mu = \frac{|\mathcal{Y}|}{\lambda - \log n}$ words $G_1, G_2, \ldots, G_\mu$. Finally, we store in the encrypted index the compressed list $\Gamma(w) = \{G_1, \ldots, G_\mu\}$. Notice that in the actual construction we store for each word its rank i by attaching $\log n$ bits so that decompression works correctly. Thus, each compressed word will have size λ bits. Our approach is described in Figure 4.1.

4.2.1 Choosing Compression Algorithms

We note that there are more than 20 different algorithms for bitmap/inverted list compression — see [57], and therefore identifying the most suitable compression algorithm is a challenging task. In our approach we selected two compression

algorithms EWAH and FastPfor that take into consideration the specialized structure of SE/OSE, i.e. how we can efficiently compress $|\mathcal{D}(w)|$ uniformly distributed document identifiers for each keyword w. We choose our compression algorithm for each keyword list in a greedy fashion: Find the best compression algorithm for each keyword list $\mathcal{D}(w)$ individually, from a set of compression algorithms $\mathcal{C}$ and store some extra metadata to denote which compression algorithm was used. Below, we explain the reasons we chose EWAH and FastPfor and in which ranges we expect that each compression algorithm will be used in practice.

We chose EWAH for keyword lists of very large size. In particular, we modify the EWAH algorithm to yield compressed words of size $\phi = \lambda - \log n$, instead of the original 32 bits. Setting the compressed words to have size ϕ bits means that the maximum number of required compressed words will be $O(\frac{n}{\phi})$. Intuitively, we expect that we can benefit from this algorithm only when $|\mathcal{D}(w)| > \frac{n}{\phi}$. If $|\mathcal{D}(w)| < \frac{n}{\phi}$ then with high probability the expected load of each compressed word will be ≤ 1, since the distribution of document identifiers is uniform. In the latter case, we represent each document identifier with $O(\lambda)$ bits, leading to no compression. EWAH achieves savings only if $\mathcal{D}(w) > n/\phi$. For example, in the extreme case that $|\mathcal{D}(w)| = n$, then EWAH will compress all the document identifiers in exactly one compressed keyword with λ bits; in this case EWAH achieves the best possible compression ratio.

We chose FastPfor for keyword lists with small, medium and large sizes. In the extreme case, that $|\mathcal{D}(w)| = n$, FastPfor will compress keyword w using approximately $O(n)$ bits, since it will require at least 1 bit per delta. Thus, we expect that

EWAH will be superior for very large keyword-lists, i.e. $|\mathcal{D}(w)| > c_1 \cdot n/\lambda$ (for some constant c_1); FastPfor will handle the remaining keyword-lists. However, there is not a clear separation of the ranges where each of the above compression algorithms will be better and so we follow a greedy selection as we described above.

In the case that the compressed keyword-lists have size greater than $\log n * |\mathcal{D}(w)|$ bits, we use the original uncompressed representation. Notice that the greedy selection is a viable solution and does not significantly affect the Setup time, since the encryption cost is the dominant factor.

4.2.2 Our SE Construction

Our main SE construction is shown in Figure 4.2. Note the random document identifier reassignment (Line 2 of the Setup algorithm), compression (Line 4 of the Setup algorithm) and partitioning (Line 5 of the Setup algorithm). In particular, Setup works as follows. After parsing the input index $\mathcal{D}$ the algorithm compresses each keyword list $\mathcal{D}(w_i)$ individually by greedily selecting the best compression method from the set of lossless compression algorithms $\mathcal{C}$ (described in Section 4.2.1). Then, Setup performs partitioning as described above to obtain the index Γ, which is padded with up to N entries and encrypted using any SE scheme as a black-box.

As shown in Figure 4.2 the algorithms KeyGen and Search perform only calls to the blackbox algorithms SE.KeyGen and SE.Search, respectively.

The Search algorithm is applied to an input keyword w, in order to retrieve the list $\{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$ and decompresses the bit string $G_1||G_2||\ldots||G_\mu$ using

$k \leftarrow \mathsf{KeyGen}(1^\lambda)$

1: $k \leftarrow \mathsf{SE.KeyGen}(1^\lambda)$.
2: **return** k.

$(st_\mathsf{C}, \mathcal{I}) \leftarrow \mathsf{Setup}(k, \mathcal{D})$

1: Set $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$. Let n be the number of documents ($n \leq N$).
2: Reassign document identifiers using a random permutation $p : [n] \rightarrow [n]$ (i.e., document i becomes document $p(i)$).
3: **for** each $w \in \mathbf{W}$ **do**
4: $(\mathcal{Y}, c) \leftarrow$ **COMPRESS**$(\mathcal{D}(w), \mathcal{C})$. ▷ $\mathcal{C}$ is a set of compression algorithms; c are the bits encoding this choice.
5: Write $\mathcal{Y}$ as $G_1||G_2 \ldots ||G_\mu$ where G_i is a bit string of $\lambda - \lceil \log n \rceil$ bits (pad the last bit string if needed).
6: Set $\Gamma(w) = \{c, 1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
7: Pad Γ to have N entries. ▷ We insert a dummy keyword which contains the necessary number of dummy values.[a]
8: $\mathcal{I} \leftarrow \mathsf{SE.Setup}(k, \Gamma)$.
9: Set st_C to include k.
10: **return** $(st_\mathsf{C}, \mathcal{I})$.

$(\mathcal{X}, st_\mathsf{C}, \mathcal{I}) \leftrightarrow \mathsf{Search}(st_\mathsf{C}, w, \mathcal{I})$

1: $(c, result) \leftarrow \mathsf{SE.Search}(st_\mathcal{C}, w, \mathcal{I})$.
2: Write $result$ as $\{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
3: $\mathcal{X} \leftarrow$ **DECOMPRESS**$(G_1||G_2|| \ldots ||G_\mu, \mathcal{C}, c)$.
4: **return** $(\mathcal{X}, st_\mathsf{C}, \mathcal{I})$.

[a] This dummy keyword is never returned as part of any query.

Figure 4.2: Our more efficient SE construction using any SE and a set $\mathcal{C}$ of compression algorithms as black-box.

the same compression algorithm $c \in \mathcal{C}$ that was used for compression. Finally, it outputs the real document identifiers $\{id_1, id_2, \ldots\}$.

Note that the utilized black-box SE scheme *must leak the actual access pattern* (e.g., [1, 32][2]), since otherwise our construction is not correct. This is because randomizing the identifiers in the black-box SE scheme would cause the decompression

[2] In the literature of SE schemes for the keyword search problem, some of the prior works [30, 31] focus for simplicity only on retrieving the document identifiers of a queried keyword w and not on getting the actual documents. These schemes do not leak the actual access pattern, but only its size. However, it is easy to extend these schemes, such that they return the actual documents and leak the actual access pattern.

$\underline{(\mathcal{I}, st_{\mathcal{S}}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))}$

1: $(N, n) \leftarrow \mathcal{L}_{\mathsf{SETUP}}(\mathcal{D})$.
2: $(\mathcal{I}, st) \leftarrow \mathrm{SE}.\mathsf{SimSetup}(N)$.
3: Let $A = \{1, 2, \ldots, n\}$ be the set of document identifiers.
4: Let Previous and Access be empty hash tables.
5: **return** $(\mathcal{I}, (st, A, \mathsf{Previous}, \mathsf{Access}))$.

$\underline{(X, st_{\mathcal{S}}, \mathcal{I}) \leftarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w), \mathcal{I})}$

1: Parse $st_{\mathcal{S}}$ as $(st, A, \mathsf{Previous}, \mathsf{Access})$.
2: Let $(id(w), R_1, R_2, \ldots, R_s) \leftarrow \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w)$.
3: **if** $\mathsf{Previous}.\mathbf{get}(id(w)) \neq \mathsf{null}$ **then**
4: **return** $(\mathsf{Previous}.\mathbf{get}(id(w)), st_{\mathcal{S}}, \mathcal{I})$.
5: **else**
6: **for** $i = 1, \ldots, s$ **do**
7: **if** $\mathsf{Access}.\mathbf{get}(R_i) = \mathsf{null}$ **then**
8: Pick id_i uniformly at random from A and set $A = A - \{id_i\}$.
9: $\mathsf{Access}.\mathbf{put}(R_i, id_i)$.
10: **else**
11: $id_i \leftarrow \mathsf{Access}.\mathbf{get}(R_i)$.
12: $(\mathcal{Y}, c) \leftarrow \mathbf{COMPRESS}(id_1||id_2||\ldots||id_s, \mathcal{C})$.
13: Write $\mathcal{Y}$ as $G_1||G_2\ldots||G_\mu$ where G_i is a bit string of $\lambda - \lceil \log N \rceil$ bits (pad the last bit string if needed).
14: Set $\Gamma(w) = \{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
15: $(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I}) \leftarrow \mathrm{SE}.\mathsf{SimSearch}(st_{\mathcal{S}}, \Gamma(w), \mathcal{I})$.
16: Parse $\mathcal{X}$ as $(c, result)$ and compute $\mathcal{X}' \leftarrow \mathbf{DECOMPRESS}(result, \mathcal{C}, c)$.
17: $\mathsf{Previous}.\mathbf{put}(id(w), \mathcal{X}')$.
18: Update $st_{\mathcal{S}}$ with the new values of $(st, A, \mathsf{Previous}, \mathsf{Access})$.
19: **return** $(\mathcal{X}', st_{\mathcal{S}}, \mathcal{I})$.

Figure 4.3: Simulator algorithms SimSetup and SimSearch for SE (keyword search problem)

algorithm to produce garbage. In addition, in the keyword search problem it is not necessary for the used black-box SE to leak both N and n; there are SE schemes that leak only N. However, our construction additionally leaks n—this allows us to define the domain from which we draw the document identifiers.

We will now prove security of our SE-K construction, assuming the black-box SE scheme we use is $(\mathcal{L}_1, \mathcal{L}_2)$-secure where $\mathcal{L}_1$ leaks only the size of the index N (but

not the number of documents n) and $\mathcal{L}_2$ leaks the search pattern and access pattern. We provide the proof for the keyword search problem; proofs for the database search problem are easily derived from the proof we present.

Theorem 2 *Let $\mathcal{L}_{\mathsf{SETUP}}$, $\mathcal{L}_{\mathsf{SEARCH}}$ be the leakages defined in Section 4.1 for the SE-K construction. If the SE scheme used as a black-box in our construction of Figure 4.2 is $(\mathcal{L}_1, \mathcal{L}_2)$-secure according to Definition 1, then our SE-K construction of Figure 4.2 is ($\mathcal{L}_{\mathsf{SETUP}}$, $\mathcal{L}_{\mathsf{SEARCH}}$)-secure.*

Proof 3 *Our black-box SE scheme is $(\mathcal{L}_1, \mathcal{L}_2)$-secure, we use its* SE.SimSetup *and* SE.SimSearch *algorithms.*

Our simulator $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$ *and* $\mathsf{SimSearch}(\mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w))$ *is described in Figure 4.3. For the first part of the proof, we must show that no PPT algorithm* Dist *can distinguish, with more than negligible probability, between the index $\mathcal{I}_{real}$ output by* $\mathsf{Setup}(k, \mathcal{D})$ *and the index $\mathcal{I}_{ideal}$ output by* $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$*. This is because both $\mathcal{I}_{real}$ and $\mathcal{I}_{ideal}$ have the same number of entries and the black-box SE is $(\mathcal{L}_1, \mathcal{L}_2)$-secure.*

For the second part of the proof we need to prove that Dist *cannot distinguish between the outputs of* $\mathsf{Search}(k, w)$ *and the output of* $\mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w))$*.*

First, both Search *and* SimSearch *produce the same messages, i.e. tokens and results, for the same repeated keywords.* SimSearch *uses the search pattern leakage* Previous *included in $st_{\mathcal{S}}$. Second, for a keyword w that has not been queried before, it is enough to show that the distribution of $\Gamma(w)$ in Line 6 of* Setup *and the distribution of $\Gamma(w)$ in Line 14 of* SimSearch *are identical. If so, the security will follow from*

the existence of a simulator of the black-box secure SE scheme. It is easy to see that the aforementioned distributions are identical. In the real game, the document identifiers that are compressed are chosen uniformly at random due to the random permutation at Line 2, and in the SimSearch *algorithm the identifiers are again picked uniformly at random from A at Line 8, every time a new keyword comes in. The simulator also correctly simulates the overlapped document identifiers between different queries using its state* $st_{\mathcal{C}}$ *and the access pattern leakage—see Lines 2 and 11.*

Choosing the black-box SE. Our solution can achieve a high degree of parallelism, good locality trade-offs and dynamism, when selecting the SE black-box scheme to be the optimal read efficiency scheme proposed in [32]. According to [50], the latter scheme achieves optimal space and locality trade-offs since it matches a lower-bound for schemes with optimal read efficiency. The composition of our scheme with the above provides very efficient search time for both in-memory and external memory settings in the standard SE leakage profile. The schemes of [32] with non-optimal read efficiency have different leakage profile; however our scheme can use them as a black-box inheriting all the different trade-offs that they provide, but in that case our scheme will inherit also their leakage profile.

4.2.3 Our OSE Construction

Our OSE-K/OSE-D construction is shown in Figure 4.4 and is based on modifying the SE construction presented in the previous section. The main difference is

that instead of using a SE scheme as a black-box, we now use an Oblivious RAM scheme for that purpose. We summarize these modifications.

- The Setup algorithm uses Lines 4 to compute the optimal worst-case number of compressed words μ_0, i.e., it computes for each used compression algorithm the worst-case required number of compressed words (for a given n and $|\mathcal{D}(w)|$) and chooses the most efficient one (we will explain the intuition behind this point below). In Line 8, list $\Gamma(w)$ is padded to contain exactly μ_0 compressed words. In Line 9, Γ is padded to have $2 \cdot N$ entries. In Line 10, Setup computes the encrypted index using OramInitialize, i.e., $(st_{\mathcal{C}}, \mathcal{I}) \leftarrow \text{OramInitialize}(k, \mathcal{D})$ and in Line 11 it outputs $(st_{\mathsf{C}}, \mathcal{I})$.

- The Search algorithm calls OramAccess as many times as necessary to receive the entire compressed result, i.e., $\forall i \in [0, \mu + 1)$ we call

$$(\mathcal{X}, st'_{\mathcal{C}}, \mathcal{I}') \leftrightarrow \text{OramAccess}(st_{\mathcal{C}}, st_{\mathcal{C}}.pos(G||i), \mathcal{I}) \,.$$

  The client's state $st_{\mathcal{C}}$, comprises all the information that the client needs to know in order to retrieve each G_i, i.e., a mapping indicating that G_i is stored in index j (this mapping is called *position map* and we denote it as $st_{\mathcal{C}}.pos(G_i)$—we will further explain the notion of position map below).

Important observation concerning security. We observe that in the SE construction (Figure 4.2) a keyword-list of $|\mathcal{D}(w)|$ size may be compressed into μ compressed words in one execution, while in another execution it may be compressed

$k \leftarrow \mathsf{KeyGen}(1^\lambda)$

1: **return** $k \leftarrow^{\$} \{0,1\}^\lambda$.

$(st_{\mathsf{C}}, \mathcal{I}) \leftarrow \mathsf{Setup}(k, \mathcal{D})$

1: Set $N = |\{\mathcal{D}(w)\}_{w \in \mathbf{W}}|$. Let n be the number of documents ($n \leq N$).
2: Reassign document identifiers based on a random permutation $p : [n] \to [n]$.
3: **for** each $w \in \mathbf{W}$ **do**
4: Compute $(\mu_0, c) \leftarrow$ **WORST-COMPRESSION**$(|\mathcal{D}(w)|, n, \mathcal{C})$.
5: $(\mathcal{Y}, c) \leftarrow$ **COMPRESS**$(\mathcal{D}(w), c)$.
6: Write $\mathcal{Y}$ as $G_1||G_2 \ldots ||G_\mu$ where G_i is a bit string of $\lambda - \lceil \log n \rceil$ bits (pad the last bit string if needed).
7: Set $\Gamma(w) = \{c, 1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
8: Pad $\Gamma(w)$ to have exactly μ_0 bit-strings of the form $x||G_x$.
9: Pad Γ to have $2 \cdot N$ entries. ▷ We insert a dummy keyword which contains the necessary number of dummy values.[a]
10: $(st_{\mathcal{C}}, \mathcal{I}) \leftarrow \textsc{OramInitialize}(k, \Gamma)$.
11: **return** $(st_{\mathcal{C}}, \mathcal{I})$.

$(\mathcal{X}, st'_{\mathcal{C}}, \mathcal{I}') \leftrightarrow \mathsf{Search}(st_{\mathcal{C}}, w, \mathcal{I})$

1: $i = 0$.
2: **while** $G_i \neq \perp$ **do**
3: $(G_i, st'_{\mathcal{C}}, \mathcal{I}') \leftrightarrow \textsc{OramAccess}(st_{\mathcal{C}}, st_{\mathcal{C}}.pos(w||i), \mathcal{I})$.
4: $\mathcal{I} \leftarrow \mathcal{I}'$, $st_{\mathcal{C}} \leftarrow st'_{\mathcal{C}}$, $i \leftarrow i + 1$.
Write *result* as $\{1||G_1, 2||G_2, \ldots, \mu||G_\mu\}$.
5: $\mathcal{X} \leftarrow$ **DECOMPRESS**$(G_1||G_2|| \ldots ||G_\mu, \mathcal{C}, c)$.
6: **return** $(\mathcal{X}, st'_{\mathcal{C}}, \mathcal{I}')$.

$(\mathcal{I}, st_{\mathcal{S}}) \leftarrow \mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$

1: Let $(N, n) \leftarrow \mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}, w)$ and $(\mathcal{I}, st) \leftarrow \textsc{SimOramInitialize}(2 \cdot N)$.
2: **return** $(\mathcal{I}, st)$.

$(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I}) \leftrightarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w), \mathcal{I})$

1: Let $s \leftarrow \mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w)$, $(N, n) \leftarrow \mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}, w)$.
2: Compute $(\mu_0, c) \leftarrow$ **WORST-COMPRESSION**$(s, n, \mathcal{C})$.
3: **for** $i = 1, \ldots, \mu_0$ **do**
4: Pick a random index ind.
5: Compute $(\mathcal{X}, st, \mathcal{I}') \leftrightarrow \textsc{SimOramAccess}(st_{\mathcal{S}}, \mathsf{ind}, \mathcal{I})$.
6: Set $\mathcal{I} \leftarrow \mathcal{I}'$ and update $st_{\mathcal{S}}$ with the new values of st.
7: **return** $(\mathcal{X}, st_{\mathcal{S}}, \mathcal{I})$.

[a] This dummy keyword is never returned as part of any query.

Figure 4.4: Our OSE-K construction and the simulator algorithms $\mathsf{SimSetup}$ and $\mathsf{SimSearch}$ using an Oblivious RAM and a set $\mathcal{C}$ of compression algorithms as black-box.

into $\mu + 1$ compressed words. However, as we proved this does not induce any security issues, since in both cases the distributions of the document identifiers are the same. Therefore, even if the adversary queries the same keyword multiple times, the simulator will simulate the query only the first time and for any subsequent execution of the same query she will use the search pattern leakage to return the previously chosen result (see Line 11 of Figure 4.3). A very important difference between the SE and OSE constructions is that the latter does not leak the search pattern, i.e., whether two encrypted search queries are the same. Let us consider the case of an adversary querying the same keyword w_1 multiple times in our OSE construction. In that case Setup will **always** produce the same number of compressed words μ, while SimSetup might yield a different number of compressed words in every execution.

In order to address the aforementioned problem, it is required that all keyword lists of the same size s to have the same number of compressed words. To achieve this, **WORST-COMPRESSION** (in Line 4) computes for each $c \in \mathcal{C}$ the worst-case compression (given n, s and $\mathcal{C}$) and returns the best algorithm c, and the worst-case number of compressed words μ_0 for c. Now, we first compress the list $\mathcal{D}(w)$ as before (see Lines 5-7) and then pad it to size μ_0 (Line 8).

We note here that computing μ_0 is easy for some algorithms, e.g., WAH, EWAH but for other algorithms, such FastPfor, it is not. It is also possible that the worst-case compression for some algorithms to be very close to the uncompressed size, e.g. VB compression algorithm described in [58]. For compression algorithms in which computing the worst-case compression is either not viable in practice or

the compression ratio is close to 1, we use an alternative methodology. We choose for each n and s a predefined μ_0, and we store the overflowed lists $(\mu - \mu_0)$ in a local stash in the client side. It is a good practice to choose μ_0 to be the expected number of compressed words (for a given n and s).

We will now prove the security of our OSE-K construction, assuming the black-box ORAM we use is secure and assuming that $\mu_0 > \mu$ always hold for Lines 4-8.

Theorem 3 *Let $\mathcal{L}_{\mathsf{SETUP}}$, $\mathcal{L}_{\mathsf{SEARCH}}$ be the leakages defined in Section 4.1 for OSE-K. If the ORAM scheme used as a black-box in the construction of Figure 4.2 is secure, then our OSE-K construction of Figure 4.2 is ($\mathcal{L}_{\mathsf{SETUP}}$, $\mathcal{L}_{\mathsf{SEARCH}}$) secure.*

Proof 4 *The deployed black-box ORAM scheme is considered to be secure, so our proof uses its* SimOramInitialize *and* SimOramAccess. *Our simulators* $\mathsf{SimSetup}(\mathcal{L}_{\mathsf{SETUP}}(\mathcal{D}))$ *and* $\mathsf{SimSearch}(\mathcal{L}_{\mathsf{SEARCH}}(\mathcal{D}, w))$ *are shown in Figure 4.4.*

For the first part of the proof, we show that no PPT Dist *algorithm exists that can distinguish, with more than negligible probability, between the index $\mathcal{I}_{real}$ and the index $\mathcal{I}_{ideal}$ since both have the same number of entries and the black-box ORAM scheme is secure. For the second part of the proof, we show that no PPT algorithm* Dist *exists that can distinguish, with more than negligible probability, between the index* Search *and the index* SimSearch*, for the following reasons; (i) both* Search *and* SimSearch *produce indistinguishable messages, (ii) in both cases* Dist *observes the same number of ORAM accesses, (iii) ORAM being secure implies that* Search *and* SimSearch *are indistinguishable with non-negligible probability.*

Choosing the ORAM black-box. Our OSE schemes can use any secure ORAM

as a black-box. For instance, we propose using any Square Root ORAM, hierarchical-based ORAM or tree-based ORAM. The main efficiency metrics for an ORAM scheme are: (i) Amortized overhead, (ii) Worst-Case Overhead, (iii) Storage, (iv) Client Storage. In our solution we assume for simplicity reasons that the client locally stores a position map, i.e., a data structure that maintains mappings of specific keyword,id pairs to their currently stored indexes j in the ORAM. Different families of ORAM schemes stores a position map in different ways. For instance, PathORAM proposes a solution that increases the overhead and recursively outsources the position map in an oblivious manner. This work we suggest that any tree-based approach with the minimum worst-case overhead even if it stores the position map locally on the client, such as the non-recursive PathORAM, to be a good candidate for constituting the ORAM black-box. We can outsource the position maps using the notion of oblivious data structures described in [41] without increasing the worst case overhead (we create a single-linked list connecting all G_i together, each G_i will store the position of G_{i+1}, we store G_1 in an oblivious data structure and the remaining G_i in PathORAM). Creating a practical OSE scheme combining PathORAM with the idea of oblivious data structure requires in total, $O(N)$ space, $O(\log^2 N)$ worst-case overhead for result sizes smaller than $O(\log^2 N)$ ids, $O(\log N)$ worst-case overhead for result sizes greater than $O(\log^2 N)$ and client storage of $O(\log^2 N) \cdot \omega(1)$ ids. We omit providing further details on combining PathORAM with oblivious data structures, as our OSE construction is generic and can improve the search performance of an OSE scheme using any ORAM as black-box. We further refer the reader to the recent work of Chang et al. [59] for a comprehensive

evaluation of various ORAM protocols.

4.3 Experiments

In this section we experimentally evaluate the performance of our proposed schemes. We call the SE construction of section 4.2.2 as microSE and the OSE construction of section 4.2.3 as microOSE. We select the SE black-box scheme to be the basic construction of Cash et al. [1] as it is the state-of-the-art in-memory SE scheme with linear size encrypted index (it requires N encrypted entries) and we refer to it as *PiBas*. We did not choose the scheme with optimal read efficiency of Demertzis and Papamanthou [32] since it requires sN space; for $s = 1$ both schemes have the same performance; for $s > 1$ the optimal read efficient scheme of [32] outperforms *PiBas* at the cost of more space. Furthermore, we denote by "microSE(*PiBas*)" that microSE uses *PiBas* as a black-box. We choose *PathORAM* [5] to be the black-box ORAM scheme for microOSE (microOSE(*PathORAM*)) and for simplicity we store the position map locally.

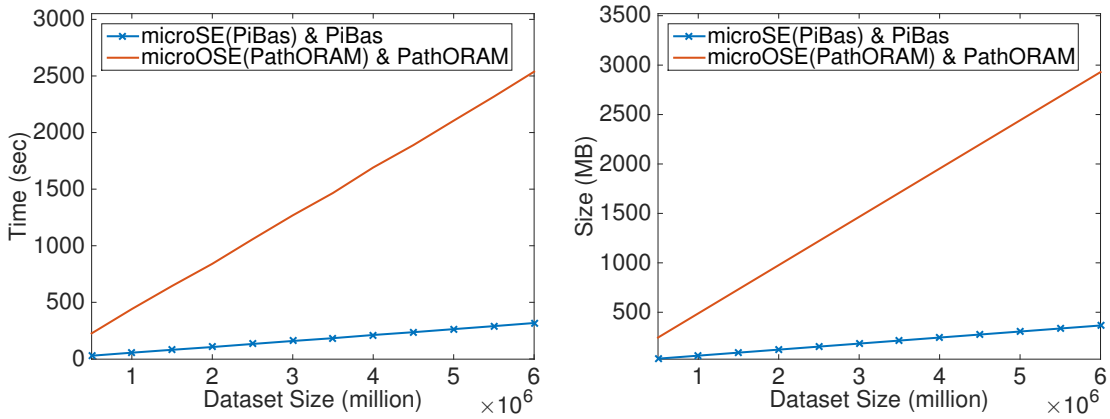
We evaluate the performance of microSE(*PiBas*) and compare it to one of the original *PiBas* scheme in order to illustrate the superiority of microSE; similarly we compare microOSE(*PathORAM*) to *PathORAM*.

4.3.1 Setup

We carried out the implementation of our schemes, *PiBas* and *PathORAM* in Java. Our experiments were conducted on a 64-bit machine with an Intel Xeon

E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto library and the bouncy castle library [53] for the cryptographic operations[3]. In particular, for the PRF and randomized symmetric encryption implementations we used HMAC-SHA-256 and AES128-CBC, respectively, for encryption. The compression algorithms that we use are *FastPfor*[60] and *EWAH* [61] (with compressed keywords of size λ bits). We consider the following two datasets in our experimental setting. The first dataset is a real dataset [46] consisting of $6,123,276$ tuples with 22 attributes of reported incidents of crime in Chicago [46]. This is a typical database table, which does not have intersections between the keywords (database search). We consider the first query attribute to be the *location description* attribute which is an attribute following a skewed distribution containing 173 distinct keywords (this is the x-axis in Figures 4.6(a),and 4.6(b)). Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency has $1,631,721$ records. We also consider the attribute *date* that does not follow a skewed distribution, in order to show the difference between a skewed and a "non-skewed" distribution in the database search case. The *date* attribute contains $58,404$ distinct keywords (this is the x-axis in Figures 4.7(a) and 4.7(b)). Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency has $14,564$ records. In Figure 4.9(a). we provide the mean and best compression ratio for all the 22 attributes.

[3]We highlight that our Java implementation does not use *hardware supported* cryptographic operations. However, this does not affect our conclusions on the superiority of our proposed constructions. The use of hardware supported cryptographic operations will drastically improve both, the original constructions of *PiBas* and *PathORAM*, as well as our own microSE(*PiBas*) and microOSE(*Path*) based constructions, thus maintaining the exact same "speed-up".



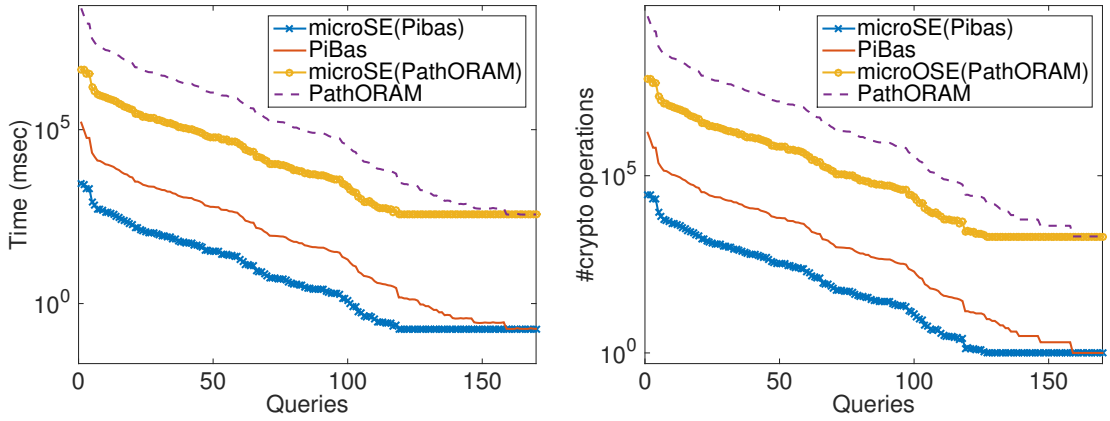
(a) Construction time (b) Index size

Figure 4.5: Index costs

For our second dataset, we use the Enron email dataset [47], which consists of $30,109$ emails from the "sent mail" folder of 150 employees of the Enron corporation that were sent between $2000-2002$. We extracted keywords from this dataset. The words were first stemmed using the standard porter stemming algorithm [62], and then we removed 200 stop words.This dataset contains $76,577$ distinct keywords (this is the x-axis in Figures 4.8(a) and 4.8(b)). Among these keywords the one with minimum frequency contains 1 id, while the one with maximum has $24,642$.

4.3.2 microSE/microOSE Evaluation

Index Costs. In the first set of experiments we evaluate the required index size and construction time of our scheme for different dataset sizes N. The results are shown in Figure 4.5. The construction time includes the I/O cost of transferring the dataset from the disk to the main memory, and the index size represents only the size of the encrypted index. Moreover, we partition the initial dataset into 12 sets of 500K tuples each, chosen uniformly at random from the entire dataset. Then, we



(a) Search Time

(b) # PRF evaluations

Figure 4.6: Search costs - Crime Dataset (Location attribute)

begin with the first partition and consider the other partitions in each step in order to represent the construction time (Figure 4.5(a)) and the index size (Figure 4.5(b)), as the size of the input gradually increases. Since we perform the same amount of work for every partition while building up the index, the storage and construction time required is linear in the number of partitions (or input size). Figure 4.5 reflects this observation. We observe that both microSE($PiBas$) and $PiBas$ have the same index costs, since microSE($PiBas$) performs padding to have exactly the same encrypted index size with $PiBas$, the same applies for microOSE($PathORAM$) and $PathORAM$. We highlight that the padding in the case of microSE affects only the setup costs since the inserted dummy records are never returned as part of any query, while in the case of microOSE it may slightly affect the query costs (depending on how we handle the overflowed lists), but the returned compressed response cannot exceed the uncompressed.

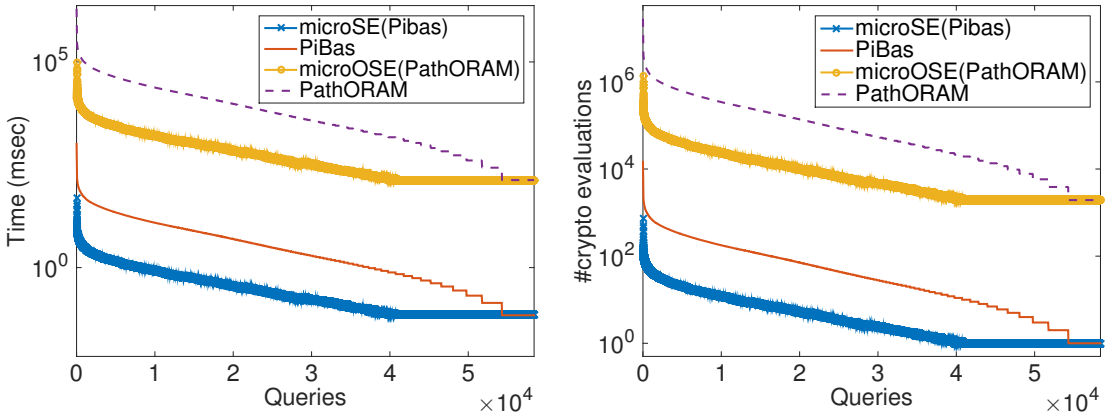
Search Cost. In this set of experiments, we illustrate the total time required by the server to retrieve and find the tuple-ids or document-ids for each query. For

visualization purposes, we sort the queries based on their result size in descending order and we query each of them, i.e. x-axis for value x=0 depicts the query with the largest result size. In Figure 4.6(a) we observe the search time and in Figure 4.6(b) the number of cryptographic operations for the *location description* attribute both for microSE and microOSE. Similarly, in Figure 4.7(a) we observe the search time and in Figure 4.7(b) the number of cryptographic operations for the *date* attribute. In the case of microOSE, we calculate a specific μ_0 for a given size, by estimating heuristically its expected value for a given $n, |\mathcal{D}(w)|$ and we store the overflowed lists in a local stash γ on the client side. We experimentally observed that local stash γ was always smaller than the stash of *PathORAM*.

The maximum speed-up for the *location description* attribute is **62**× both for microSE and microOSE, while for the date attribute the corresponding number is **21**×. The *location description* attribute presents a more skewed distribution, as it contains high-frequency keywords. Note that more tuple-ids per keyword lead to a better compression ratio. microSE and microOSE have the same performance because in both cases they take advantage only of the size of each query.

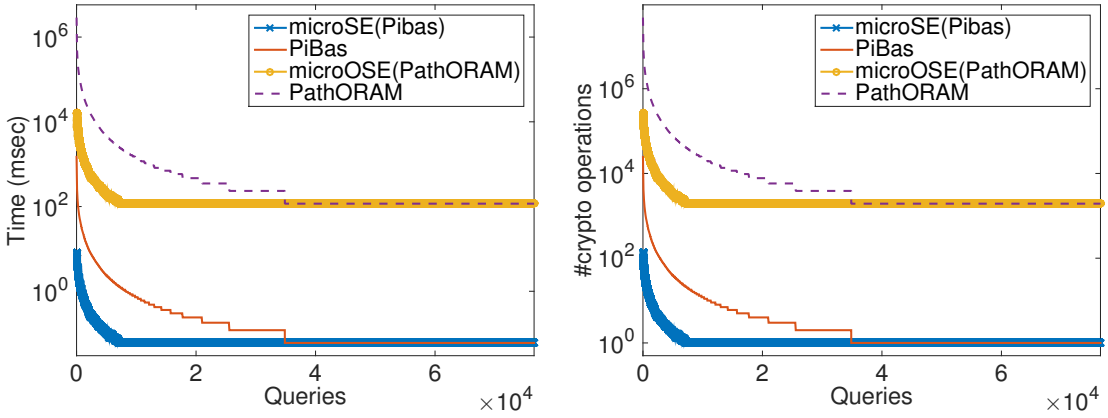
In Figure 4.8(a), we observe the search time and in Figure 4.8(b) the number of PRF evaluations for the Enron dataset. In the case of SE we achieve up to **188**× speed-up, while in the case of OSE the speed-up was similar since in both cases the compression takes advantage of the number of documents and the size of each query.

In Figure 4.9(a), we use microSE for all the 22 attributes of the Crime dataset and we report the best and the mean speed-up that we achieve. We observe that for the attributes $1, 2$ the compression ratio is 1, which means that no compression



(a) Search Time (b) # PRF evaluations

Figure 4.7: Search costs - Crime Dataset (Date attribute)

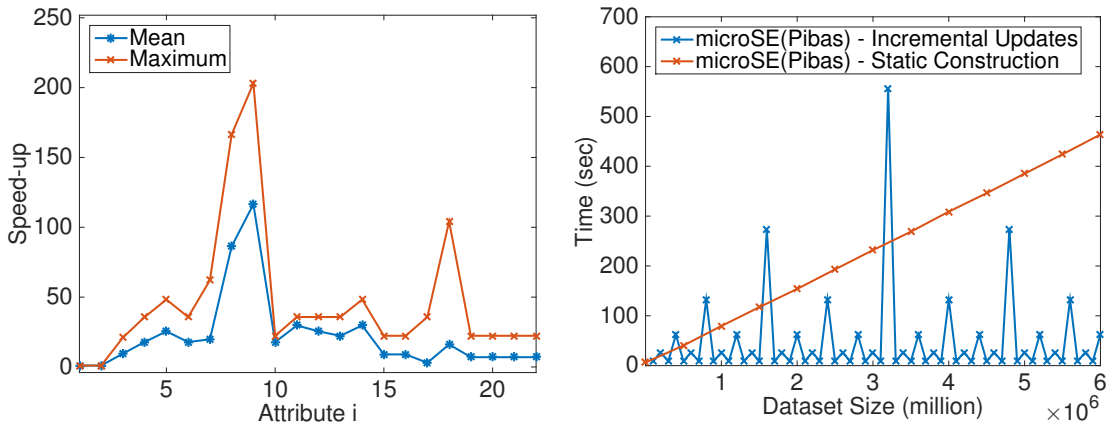


(a) Search Time (b) # PRF evaluations

Figure 4.8: Search costs - Enron Dataset

is achieved. The reason is that the first 2 attributes contain unique values, so every value has result size 1, which is the minimum number of cryptographic operation that we have to perform. We also observe that attributes $8, 9$ achieve higher compression ratio than the other attributes (up to **166**×, **203**× respectively); the reason is that these are binary attributes (true or false) and the sizes of their values are proportional to the database size (attribute 8: whether an arrest was made or not, attribute 9: whether the incident was domestic-related or not).

Dynamic costs. In this set of experiments, we consider the case of dynamic mi-



(a) Search Time Speed-up (22 attributes) (b) Update Costs

Figure 4.9: Additional Experiments (Crime Dataset)

croSE which is addressed as described in [2]. For these experiments, we fix the consolidation step s that is described in the original paper, to 2. This means that after every 2 new indexes, we initiate a consolidation phase that merges one or more indexes in order to construct a new one. The batch size is set to $100,000$ updates. Figure 4.9(b) plots the time required for dynamic microSE($Pibas$) (labeled as "Incremental Updates") to maintain the index, when considering 10Mbps network bandwidth. This experiment includes the time required for downloading, decrypting, reconstructing (merging), re-encrypting and uploading the indexes. As a reference, the plot also includes the cost required by static microSE($Pibas$) to build the same index (including the time for uploading the index), assuming that the whole dataset is made available at once (labeled as "Static Construction"). We can also use the same approach in order to extend microOSE to the dynamic setting; the update costs will follow the same pattern but they will be scaled by a constant factor.

Chapter 5: Dynamic Searchable Encryption with Small Client Storage

Recent research has focused on *dynamic searchable encryption (DSE)* schemes that can efficiently support modifications in the encrypted dataset, without the need to re-initialize the protocol. From a security perspective, developing secure DSE schemes is challenging due to the additional information that may be revealed to the server because of updates. Two relevant security notions have been proposed for dynamic SE schemes—*forward* and *backward privacy*. Forward privacy [24, 63] ensures that a new update cannot be related to previous operations (until the related keyword is searched). Besides the obvious benefit of allowing the encrypted dataset to be built "on-the-fly" (crucial for certain applications, e.g., encrypted e-mail storage starting from a new mailbox), forward privacy is essential for mitigating certain leakage-abuse attacks that depend on adversarial file injection [64].

On the other hand, backward privacy ensures that if a document containing keyword w is deleted *before* a search for w, the result of this search does not reveal anything about this document. Backward privacy is much less studied than forward privacy. It was first proposed in NDSS 2014 by Stefanov et al. [24] and formally defined recently in CCS 2017 by Bost et al. [65] who proposed three types of

backward-privacy. During a search, BP-I reveals only the identifiers of files currently containing w and when they were stored, BP-II additionally reveals the timestamps and types (insertion/deletion) of all prior updates for w, and BP-III additionally reveals for each prior deletion which insertion it canceled.

Challenge 1: DSE with small client storage. The majority of practical DSE constructions from the literature (e.g., [28, 65, 66, 67]) require the client to locally store a table that holds for every keyword in the dataset a counter a_w that counts the number of updates for w (some schemes store an additional counter for searches). This allows for very efficient schemes in practice, e.g., insertion of the entry (w, id, add) after a_w updates can be done by encrypting (w, id) and placing the ciphertext in a hash map (stored at the server) at position $F(k, (w, a_w + 1))$, where F is a pseudorandom function (updates also can contain deletes which are handled by inserting cancelation tuples). Later, to search for w the client simply looks up the value of a_w and queries the map at locations $F(k, (w, 1)), \ldots, F(k, (w, a_w)$.

With some variations, this is the basic blueprint of many existing schemes. This *local word counter* gives very efficient schemes but it has an obvious drawback: *increased client storage*. Compared to storing an inverted index for DB locally, the client needs to store a table W of unique keywords which, depending on the dataset, may be rather large. E.g., for the Enron e-mail dataset, $|DB| \approx 2.6$M and $|W| \approx 77K$, i.e., the client has to go through the trouble of deploying a DSE (and leaking information) just to reduce its local storage by 33×. When using SE to store relational database records (e.g., [32, 35, 68]) the savings can be significantly

Scheme	Computation		Communication		Search RT	Client Storage	BP
	Search	Update	Search	Update			
Moneta [65]	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^2 N)$	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^3 N)$	2	$O(1)$	I
WO+Mitra [66]	$O(a_w + \log^2 \lvert W \rvert)$	$O(\log^2 \lvert W \rvert)$	$O(a_w + \log^2 \lvert W \rvert)$	$O(\log^2 \lvert W \rvert)$	$O(\log \lvert W \rvert)$	$O(1)$	II
SD_a	$O(a_w + \log N)$	$O(\log N)(am.)$	$O(a_w + \log N)$	$O(\log N)(am.)$	1	$O(1)$	II
SD_d	$O(a_w + \log N)$	$O(\log^3 N)$	$O(a_w + \log N)$	$O(\log^3 N)$	1	$O(1)$	II
Orion [66]	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(1)$	I
Horus [66]	$O(n_w \log d_w \log N + \log^2 \lvert W \rvert)$	$O(\log^2 N)$	$O(n_w \log d_w \log N + \log^2 \lvert W \rvert)$	$O(\log^2 N)$	$O(\log N)$	$O(1)$	III
QOS	$O(n_w \log i_w + \log^2 \lvert W \rvert)$	$O(\log^3 N)$	$O(n_w \log i_w + \log^2 \lvert W \rvert)$	$O(\log^3 N)$	$O(\log \lvert W \rvert)$	$O(1)$	III

Table 5.1: Comparison of existing forward-and-backward-private DSE with small client storage. N is an upper bound for total insertions, $|W|$ = #distinct keywords. For keyword w: a_w = #updates, i_w = #insertions, d_w = #deletions, n_w = #files containing w. RT is #roundtrips for retrieving $DB(w)$. BP stands for backward privacy type (the smaller, the better) and $am.$ for amortized efficiency. The $\tilde{O}$ notation hides polylogarithmic factors. WO stands for storing search/insertion counters for each w at an oblivious map. To minimize client storage, oblivious map stashes are stored at the server and downloaded every time.

smaller, i.e., in the case of a real dataset with crime incidents in Chicago [46] (used in [32, 35]) with $|DB| \approx$ 6M tuples, 22 attributes, and $|W| \approx$ 17M, the reduction in local storage for supporting point queries for these attributes will be $5\times$ at best (similar results are observed in TPC-H benchmark [49]). In general, for relational database search many attributes may contain unique values, (e.g., every record may contain a different value) and in these cases the improvement in local storage will be negligible. The aforementioned examples clearly illustrate that in many cases storing locally a counter per word is problematic. Moreover, if we would like to support the capability to access the encrypted database from multiple devices, this approach would be especially cumbersome as it entails synchronization and state transfer among them.

Using oblivious primitives. To avoid this, previous works (e.g., [1, 28, 65, 67]) have proposed to store W at the server encrypted. This would trivially violate forward privacy, unless one uses an *oblivious map (OMAP)* [41] that hides from the server which word entry is accessed every time. One downside of this is that the construction of [41] and subsequent improvements [69] require a *logarithmic* number of rounds of interaction. The only existing DSE that avoids this is the forward-private scheme of [70] (later made backward private in [65]). However, it uses the recursive Path-ORAM construction of [5] and it relies on heavy garbled circuit computation to make it non-interactive. Therefore, its potential for adoption in practice is quite limited and it serves mostly as a feasibility result. Hence, we ask whether it is possible to design a *practical* backward-and-forward-private DSE with *small client storage* (e.g., polylog($|DB|$) or, ideally, constant) and *non-interactive* search, which

is the main motivation of our SD_a and SD_d schemes (see Table 5.1).

Challenge 2: DSE with (quasi-)optimal search. With a plaintext dataset, the n_w document identifiers of files currently containing w can be *optimally* retrieved with n_w operations. The same performance can be achieved for DSE (e.g., [28, 67]), albeit for *insertion-only* schemes (where $n_w = a_w$, the total number of updates for w). With deletion-supporting DSE n_w can be arbitrarily smaller than a_w. The only two backward-private schemes that come close to achieving this optimal performance are from [66]. At a high-level, they replace the n_w accesses necessary for retrieving the result with oblivious accesses and achieve a polylogarithmic overhead over the optimal cost (see Table 5.1 for more details). According to Definition 5, these schemes achieve *quasi-optimal* search time. However, their "black-box" use of oblivious primitives results in schemes with rather poor performance, especially due to communication cost (e.g., [66] reports Sim 1MB communication for returning just $n_w = 100$ identifiers). Therefore, we aim to develop a DSE with *quasi-optimal* search and much better *practical performance*—our QOS scheme (see Table 5.1).

Our novel DSE schemes. In this Chapter, we present novel schemes that address the above challenges as follows:

(i) We present a black-box reduction from any result-hiding static SE to a backward-and-forward private DSE. We instantiate it with [1] and call the resulting scheme SD_a. It has $O(a_w + \log N)$ search cost, where a_w denotes the total number of updates for keyword w, and $O(\log N)$ *amortized* update cost. Most importantly, SD_a is the *first* DSE with $O(1)$ permanent client storage *without*

using oblivious primitives, hence it greatly outperforms all existing schemes for searches.

(ii) During amortized updates the temporary client storage of SD_a may grow arbitrarily large (up to $O(N)$). To avoid this, we present a version with de-amortized updates called SD_d that has the same search overhead as SD_a and it outperforms state-of-the-art low-client-storage DSE schemes in many scenarios (see our experimental evaluation in Section 5.4).

(iii) Finally for delete-intensive query workloads, we present QOS, a DSE with quasi-optimal search time $O(n_w \log i_w)$ and $O(1)$ client storage that vastly outperforms existing quasi-optimal schemes during searches, where n_w denotes the number of files containing keyword w and i_w the number of insertions for w. Indeed, for large deletion percentages (approximately $40-80\%$, depending on the deployment setting) it outperforms all other schemes.

All our constructions are forward-and-backward private (BP-II for SD_a and SD_d, BP-III for QOS). In addition, our schemes are secure in the programmable random oracle model but this assumption can be removed with standard techniques without decrease in asymptotic efficiency, similar to previous works, e.g., [1, 67].

A detailed comparison with other DSE can be seen in Table 5.1 where we only focus on schemes with small client storage. We also consider WO+Mitra (WO stands for storing search/insertion counters for each w at an oblivious map), the result of combining the most efficient backward-private scheme from [66] with the

"word counter + oblivious map" approach described above (this technique can be used with other schemes, e.g., FIDES, JANUS from [65] and JANUS++ from [71], but MITRA outperforms all of them both in terms of performance and security). All schemes in Table 5.1 use OMAPs, except for SD_a; they can achieve $O(1)$ storage by storing the stashes at the server and generating keys with a PRF. One general conclusion from the table is that our schemes achieve much better search performance at the cost of increased overhead for updates. We note that this trade-off can be favorable, e.g., it seems suitable for OLAP databases and data warehouses [**?**] in which search is more crucial than the update performance.

We implemented our three schemes and compare their search, update, and storage performance with existing forward-and-backward private DSE (Section 5.4). In particular, we compare them with the best low-client-storage scheme, MITRA [66] with the word counter stored in an oblivious map, and HORUS [66], the faster quasi-optimal scheme. In terms of search time, SD_a and SD_d take less than 0.1ms for retrieving a result of 100 elements from a dataset of 1M records. Moreover, for small results, they are up to **34**× and **20**× faster than MITRA, with the added benefit of being non-interactive. Turning to quasi-optimal schemes, QOS takes 1.3ms for the same setting, vastly outperforming HORUS (**4-16531**× throughout our experiments). Where our schemes perform worse is in updates (as is evident from the asymptotic analysis in Table 5.1), e.g., for our tested cases QOS is roughly 2× slower than HORUS (with the same blowup factor for communication size), whereas MITRA is up to 21× faster than SD_d (in the worst case). All these results are for 10% deleted entries. For larger delete percentages we show that QOS has the po-

tential to become the most efficient solution. It outperforms both MITRA and SD_d after different ratios between 40-80%, depending on the number of insertions.

5.1 Dynamic Searchable Encryption (DSE)

In this section, we extend the definition of SE (provided in Section1) for the dynamic case. A *dynamic symmetric searchable encryption scheme (DSE)* $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consists of algorithm Setup, and protocols $\mathsf{Search}, \mathsf{Update}$ that are executed between a client and a server:

- $\mathsf{Setup}(\lambda)$ on input λ outputs (K, σ, EDB) where K is a secret key for the client, σ is the client's local state, and EDB is an (initially empty) encrypted database that is sent to the server. The notation $\mathsf{Setup}(\lambda, N)$ refers to a setup process that takes a parameter N for the maximum supported number of entries.

- $\mathsf{Search}(K, q, \sigma; EDB)$ is a protocol for searching the database. Here, we consider search queries for a single keyword i.e., $q = w \in \Lambda^*$. The client's output is $DB(w)$. The protocol may also modify K, σ and EDB.

- $\mathsf{Update}(K, op, w, id, \sigma; EDB)$ inserts an entry to or removes an entry from DB. Input consists of $op = add/del$, file identifier id and keyword w. The protocol may modify K, σ and EDB.

In the above, we mostly followed the description of [28, 65, 66]. Given the above API, on input the data collection the client can run Setup, followed by N

calls to Update to "populate" EDB. Assuming the scheme is forward private (see below) this leaks nothing more than running an initial setup operation on the DB. Other works [67, 72] model Update as "file" addition or deletion, where the protocol adds/removes all the relevant keywords to/from DB. This is functionally equivalent as this process can be decomposed to multiple calls of the above Update protocol.

At a high level, Σ is correct if the returned result $DB(w)$ is correct for every query (for a formal definition, see [1]). The privacy of Σ is parametrized by a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{SETUP}}, \mathcal{L}_{\mathsf{SEARCH}}, \mathcal{L}_{\mathsf{UPDATE}})$ that describes the information revealed to the server throughout the protocol execution. $\mathcal{L}_{\mathsf{SETUP}}$ refers to leakage during setup, $\mathcal{L}_{\mathsf{SEARCH}}$ during a search operation, and $\mathcal{L}_{\mathsf{UPDATE}}$ during updates. Standard search leakage types form the literature include *search pattern* that reveals which searches are related to the same w, and *access pattern* that reveals $DB(w)$ during a search for w. Note that access pattern leakage is unavoidable if the client wishes to retrieve the actual files and not just their identifiers (unless the files themselves are stored in a protected manner, e.g., Oblivious RAM). Schemes that avoid this leakage are called *result hiding*.

Informally, a secure SSE scheme with leakage $\mathcal{L}$ should reveal nothing about the database **DB** other than this leakage. This is formally captured by a standard real/ideal experiment with two games $\mathsf{Real}^{\mathsf{DSE}}$, $\mathsf{Ideal}^{\mathsf{DSE}}$ presented in Figure 5.1, following the definition of [24].

Definition 2 ([24]) *A DSE scheme Σ is adaptively-secure with respect to leakage function $\mathcal{L}$, iff for any PPT adversary Adv issuing poly(λ) queries/updates q, there*

$b \leftarrow \mathsf{Real}^{\mathsf{DSE}}(\lambda, q)$

$N \leftarrow \mathsf{Adv}(1^\lambda)$.
$(K, \sigma_0, EDB_0) \leftarrow \mathsf{Setup}(1^\lambda, N)$.
for $k = 1$ to q **do**
  $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$.
  if $type_k = search$ **then**
    $(\sigma_k, DB(w_k); EDB_k) \leftarrow \mathsf{Search}(K, w_k, \sigma_{k-1}; EDB_{k-1})$.
  else if $type_k = update$ **then**
    $(\sigma_k; EDB_k) \leftarrow \mathsf{Update}(K, add/del, (id_k, w_k), \sigma_{k-1}; EDB_{k-1})$.
Let $t_1, \ldots, t_k$ be the messages from client to server in the Search/Update protocols above.
$b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, \ldots, EDB_q, t_1, t_2, \ldots, t_q)$.
return b

$b \leftarrow \mathsf{Ideal}_{\mathcal{L}}^{\mathsf{DSE}}(\lambda, q)$

$N \leftarrow \mathsf{Adv}(1^\lambda)$.
$(st_{\mathcal{S}}, EDB_0) \leftarrow \mathsf{SimSetup}(1^\lambda, \mathcal{L}_{\mathsf{SETUP}})$.
for $k = 1$ to q **do**
  $(type_k, id_k, w_k) \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, \ldots, t_{k-1})$.
  if $type_k = search$ **then**
    $(st_{\mathcal{S}}; t_k, EDB_k) \leftarrow \mathsf{SimSearch}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{SEARCH}})(w_k); EDB_{k-1})$.
  else if $type_k = update$ **then**
    $(st_{\mathcal{S}}; t_k, EDB_k) \leftarrow \mathsf{SimUpdate}(st_{\mathcal{S}}, \mathcal{L}_{\mathsf{UPDATE}})(w_k); EDB_{k-1})$.
Let $t_1, \ldots, t_k$ be the messages from client to server in the Search/Update protocols above.
$b \leftarrow \mathsf{Adv}(1^\lambda, EDB_0, t_1, t_2, \ldots, t_q)$.
return b

Figure 5.1: Real and ideal experiments for the DSE scheme.

exists a stateful PPT simulator $\mathsf{Sim} = (\mathsf{SimSetup}, \mathsf{SimSearch}, \mathsf{SimUpdate})$ *such that*

$$\Pr[\mathsf{Real}^{\mathsf{DSE}}(\lambda, q) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{L}}^{\mathsf{DSE}}(\lambda, q) = 1]| \leq negl(\lambda).$$

Forward and backward privacy. DSE schemes with forward and backward privacy aim to control what information is revealed in relation to updates. Informally, a scheme is *forward private* if it is not possible to connect a new update to previous

operations, when it takes place. E.g., it should be impossible to tell whether an addition is for a new keyword or a previously searched one.

Definition 3 ([65]) *An $\mathcal{L}$-adaptively-secure DSE scheme that supports single-keyword additions/deletions is forward private iff the update leakage function $\mathcal{L}_{\mathsf{UPDATE}}$ can be written as: $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = \mathcal{L}'_{\mathsf{UPDATE}}(op, id)$ where $\mathcal{L}'$ is a stateless function, $op = add/del$, and id is a file identifier.*

Backward private DSE schemes limit the information that the server learns during a search for w for which some entries have been previously deleted. Ideally, the scheme should reveal nothing about these deleted entries and, at the very least, not their corresponding file identifiers [24]. Bost et al. [65] gave the first formal definition for three types of backward privacy with different leakage patterns, from Type-I which reveals the least information to Type-III which reveals the most. In order to present their definition, we need to first define some additional functions.

Let Q be a list with one entry for each operation. For searches the entry is (u, w) where u is the timestamp and w is the searched keyword. For updates it is $(u, op, (w, id))$ where $op = add/del$ and id is the modified file.

$\mathbf{TimeDB}(w) = \{(u, id) \mid (u, add, (w, id)) \in Q \wedge \forall u', (u', del, (w, id)) \notin Q\}$, is the function that returns all timestamp file-identifier pairs of keyword w that have been added to DB and have not been deleted.

$\mathbf{Updates}(w) = \{u | (u, add, (w, id)) \in Q \text{ or } (u, del, (w, id)) \in Q\}$, is the function that returns the timestamp of each insertion/deletion operation for w.

$\mathbf{DelHist}(w) = \{(u^{add}, u^{del}) \mid \exists\ id : (u^{add}, add, (w, id)) \in Q \wedge (u^{del}, del, (w, id)) \in$

$Q\}$, is the function that returns for each deletion timestamp the timestamp of the corresponding insertion it cancels.

Using the above functions, backward privacy is defined as follows.

Definition 4 ([65]) *An $\mathcal{L}$-adaptively-secure SSE scheme has backward privacy:*

- ***BP-I (BP with insertion pattern):*** *iff* $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = \mathcal{L}'(op)$ *and* $\mathcal{L}_{\mathsf{SEARCH}}(w) = \mathcal{L}''(\boldsymbol{TimeDB}(w), a_w)$,
- ***BP-II (BP with update pattern):*** *iff* $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = \mathcal{L}'(op, w)$ *and* $\mathcal{L}_{\mathsf{SEARCH}}(w) = \mathcal{L}''(\boldsymbol{TimeDB}(w), \boldsymbol{Updates}(w))$,
- ***BP-III (weak BP):*** *iff* $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = \mathcal{L}'(op, w)$ *and* $\mathcal{L}_{\mathsf{SEARCH}}(w) = \mathcal{L}''(\boldsymbol{TimeDB}(w), \boldsymbol{DelHist}(w))$,

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions. We stress that the above definitions (even BP-I) reveal the files currently containing w due to $\boldsymbol{TimeDB}(w)$—this is in order to account for the leakage from retrieving the actual files. One could define an even stronger definition that avoids this leakage (in practice this could be achieved by using oblivious storage, or when limited to applications that look to return just the identifiers and not the files). None of our constructions explicitly leaks $\boldsymbol{TimeDB}(w)$; indeed we never use it in our proofs for simulation.

DSE with optimal search time. The majority of existing DSE schemes adopt the approach of "storing" deletions as regular entries. During searches, they are used to filter out which insertion entries have been removed. This approach implies that the search cost will be $\Omega(a_w)$, i.e., linear in the total number of total updates for w, as

opposed to the optimal cost $O(n_w)$, linear in the number of files currently containing w. Notable exceptions to these are the construction of Stefanov et al. [24] (which, however, is not backward private) and two constructions from the recent work of Ghareh Chamani et al. [66] which have quasi-optimal search time according to the following definition.

Definition 5 ([66]) *A DSE scheme Σ has* optimal *(resp.* quasi-optimal*) search time, if the asymptotic complexity of Search is $O(n_w)$ (resp. $O(n_w \cdot \text{polylog}(N))$).*

5.2 From Static to Dynamic Schemes

5.2.1 Amortized construction

Our starting point is a static, result-hiding searchable encryption scheme SE, which we modify to store triplets of the form (w, id, op) (instead of the standard w, id), where $op = add/del$. The main idea behind our DSE construction called SD_a (Figure 5.3), is to organize N (without loss of generality, let N be a power-of-two) updates into a collection of $\log N$ independent encrypted indexes $EDB_0, \ldots, EDB_{\log(N-1)}$ for sizes $2^0, \ldots, 2^{\log(N-1)}$, each one created with a separate invocation of $\mathsf{SE.Setup}$ with a fresh key.

Initially, all EDB_i are empty. For the first update the client sets up an encrypted index for the singleton set (w, id, op) using $\mathsf{SE.Setup}$ and sends it to server who stores it as EDB_0. For future updates, let j be the smallest value for which EDB_j is empty. The server first sends to the client all EDB_i for $i < j$ and deletes them locally. The client fully decrypts them (we denote this in Figure 5.3 with

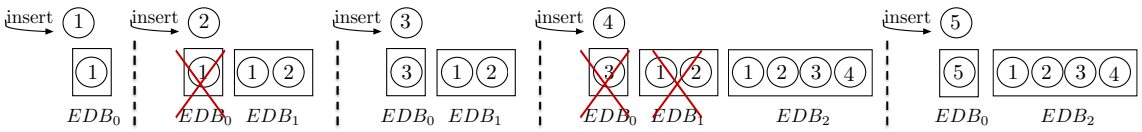


Figure 5.2: SD_a: from static to dynamic (amortized version). These are the encrypted indexes after five consecutive insertions $1-5$. Inserting element 1 requires the creation of EDB_0 which will contain element 1. Inserting element 2 requires downloading EDB_0 (to obtain element 1), creating EDB_1 which will contain elements 1 and 2, and deleting EDB_0. Searching for a keyword w requires to search all the active (non-deleted) encrypted indexes and return to the client all the individual search results.

SE.DecryptAll function) and runs SE.Setup for the union of their entries, together with the current update (w, id, op). Note that the total size of the returned EDB_i is $2^j - 1$, thus the output of SE.Setup is a new encrypted index of size 2^j; this is sent to the server who stores it as EDB_j. At all times, the client stores locally the corresponding keys and states of the different non-empty instance of SE as K and σ.

For searches, the parties run SE.Search for each (i.e., non-empty) instance of SE and return all the individual search results. Since SE is result-hiding, the client needs to do the extra work of decrypting the returned values and extracting the pairs (id, op). The final answer is the result of "filtering out" the deleted entries. Figure 5.2 illustrates the collection of the encrypted indexes after each of five consecutive inserts.

Security. We assume that the underlying SE scheme is adaptively secure. Regarding forward privacy, note that each update (w, id, op) results in running SE.Setup with a freshly chosen key. The size of the encrypted index (2^j in the above description) is fully determined by the number of previous updates, thus an update

operation can be perfectly emulated by the setup simulator of SE, even if the setup leakage of SE is just the database size. This implies that the information the server sees during updates, is independent of any previous entries in EDB (including entries about w) which gives us forward privacy. Regarding backward privacy, things are also straight-forward. Firstly, since SE is result-hiding and we store deletions as regular entries, the server does not learn the indexes of files that previously contained w. Moreover, during searches the server learns $|DB(w)|$ as well as how many result elements come from each of EDB_i. In order to simulate the second part, we only need to know when each update for w took place—this information together with the total update count so far, determines in which EDB_i each update resides. We previously defined this information as $\mathbf{Updates}(w)$), hence our scheme is BP-II.

Observe that $\mathsf{SimSearch}$ does not always need $\mathbf{Updates}(w)$ to simulate the search transcript. It suffices to know which index each update should be mapped, to according to its timestamp. The actual leakage can be much smaller—depending on the update counter upd it may be as small as $|\mathbf{Updates}(w)|$ (e.g., if $upd = 2^i$ for some $i \in \mathbb{N}$, the largest index has just been rebuilt and the previous ones are empty, hence *all* the entries for w will come from the same index and $\mathsf{SimSearch}$ does not need their individual timestamps).

Theorem 4 *Assuming* SE *is an adaptively-secure result-hiding static searchable encryption scheme,* SD_a *is an adaptively-secure DSE according to Definition 2 with* $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = \perp$ *and* $\mathcal{L}_{\mathsf{SEARCH}}(w) = \boldsymbol{Updates}(w)$.

Proof 5 *(Sketch) Building a simulator* Sim *is straight-forward, given the existence*

Let $\mathsf{SE} = (\mathsf{Setup}, \mathsf{Search}, \mathsf{DecryptAll})$ be a result-hiding, static searchable encryption scheme.

$\underline{(K, \sigma, EDB) \leftarrow \mathsf{Setup}(1^\lambda)}$

1: Set EDB to be an empty vector of indexes EDB_i
2: Set K, σ to be empty vectors

$\underline{(K, \sigma; EDB) \leftrightarrow \mathsf{Update}(K, op, w, id, \sigma; EDB)}$

Server:
1: Find the minimum j such that $EDB_j = \emptyset$
2: Send to client $EDB_0, \ldots, EDB_{j-1}$
Client:
3: Set $A \leftarrow \emptyset$
4: **for** $i = 0, \ldots, j-1$ **do**
5: $\quad A \leftarrow A \cup \mathsf{SE.DecryptAll}(K[i], \sigma[i], EDB_i)$
6: $\quad K[i] \leftarrow \perp$, $\sigma[i] \leftarrow \perp$
7: $(K[j], \sigma[j], EDB_A) \leftarrow \mathsf{SE.Setup}\ (1^\lambda, A \cup (w, id, op))$
8: Send EDB_A to server
Server:
9: Set $EDB_j \leftarrow EDB_A$
10: **for** $i = 0, \ldots, j-1$ **do**
11: $\quad$ Set $EDB_i \leftarrow \emptyset$

$\underline{DB(w) \leftrightarrow \mathsf{Search}(K, q, \sigma; EDB)}$

Client $\leftrightarrow$ Server:
1: $\mathcal{X} \leftarrow \emptyset$.
2: **for** all i such that $EDB_i \neq \emptyset$ **do**
3: $\quad$ Let $\mathcal{X}_i \leftrightarrow \mathsf{SE.Search}(K[i], q, \sigma[i]; EDB_i)$
4: $\quad \mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{X}_i$
Client:
5: Decrypt entries of $\mathcal{X}$ with K and parse them as (id, op)
6: $DB(w) \leftarrow \{id \mid (id, add) \in \mathcal{X} \wedge (id, del) \notin \mathcal{X}\}$

Figure 5.3: SD_a: from static to dynamic (amortized version).

of a simulator $\mathsf{Sim_{SE}} = \{SimSetup_{\mathsf{SE}}, SimSearch_{\mathsf{SE}}\}$. *SimSetup returns empty vector* EDB *and initializes and update counter* $upd = 0$. *During each update,* SimUpdate *computes* j *as the least significant zero bit position of* upd*, runs a new instance* $\mathsf{Sim}^{(j)}_{\mathsf{SE}} = \{SimSetup^{(j)}_{\mathsf{SE}}, SimSearch^{(j)}_{\mathsf{SE}}\}$*, executes* $SimInit^{(j)}_{\mathsf{SE}}$ *on input* 2^j*, and sends the result to the adversary. It also terminates currently running instances*

of $SimSetup^{(i)}_{\mathsf{SE}}$ *for* $i = 0, \ldots, j-1$*, and increments upd. During a search for* w*, let upd be the current update counter.* SimSearch *receives as input* $\boldsymbol{Updates}(w)$*. It then initializes values* $t_0, \ldots, t_{\lfloor \log upd \rfloor}$ *to 0. For each entry* $u \in \boldsymbol{Updates}(w)$*, it computes* i *as the index in which the update with timestamp* u *was stored (determined by* upd, u*) and increments* t_i *by one. Finally for* $j = 0, \ldots, \lfloor \log upd \rfloor$*, it runs* $SimSearch^{(j)}_{\mathsf{SE}}$ *on input* t_j*, and sends all the outputs to the adversary. Assuming* SE *is secure and result-hiding, and each instance* $\mathsf{Sim}_{\mathsf{SE}}$ *is spawned independently with fresh randomness, and given that the timestamp of an update fully determines the corresponding index structure for its entry, the transcript produced by* Sim *is indistinguishable from the messages observed by the adversary during the real protocol execution.* □

Efficiency. After N updates, SD_a consists of $\log N$ encrypted indexes, each of which is either empty or stores exactly 2^i items. Assuming SE has linear storage, SD_a has server storage $O(N)$. If SE has optimal search time, the query cost for retrieving all the updates for w is $O(a_w)$. Since there can be at most $\log N$ non-empty indexes EDB_i and a search needs to be performed in each of them, the total search time for SD_a is $O(a_w + \log N)$. Finally, after 2^j updates the client will have run SE.Setup once for size 2^j and once for 2^{j-1}, twice for 2^{j-2}, etc., all the way down to 2^{j-1} times for size one. Assuming an underlying static scheme with linear setup time, the amortized cost per update after N updates is $O(\log N)$.

One static scheme that satisfies these assumptions is the PiBas construction of [1], which we describe in Figure 5.4. Moreover, with PiBas the client has to store

Let $\mathsf{RND} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a semantically-secure encryption scheme, F be a PRF, and H be a collision-resistant hash function.

$\underline{(K, EDB) \leftarrow \mathsf{Setup}(1^\lambda, DB)}$

1: Initialize an empty map T
2: Set $(k, k') \leftarrow \mathsf{KeyGen}(1^\lambda)$
3: **for** each $w \in DB$ **do**
4: Set counter $c \leftarrow 0$
5: $(key, value) \leftarrow \mathsf{Map}(K, w, id, c)$
6: Store $(key, value)$ to T; $c{+}{+}$
7: Set $K \leftarrow (k, k')$; $EDB \leftarrow T$

$\underline{(k, k') \leftarrow \mathsf{KeyGen}(1^\lambda)}$

1: Choose random PRF key k for F
2: Set $k' \leftarrow \mathrm{RND.Enc}(1^\lambda)$

$\underline{(key, value) \leftarrow \mathsf{Map}(K, w, id, c)}$

1: $key \leftarrow H(F(k, w), c)$
2: $value \leftarrow RND.Enc(k', w, id)$

$\underline{DB(w) \leftrightarrow \mathsf{Search}(K, q; EDB)}$

Client:
1: Send $tk \leftarrow F(k, w)$ to server
Server:
2: Set $\mathcal{X} \leftarrow \emptyset$; $c \leftarrow 0$
3: **while** *true* **do**
4: Set $res \leftarrow T.\mathrm{get}(H(tk), c)$
5: **if** $res = \perp$ **then break**
6: **else** $\mathcal{X} \leftarrow \mathcal{X} \cup res$; $c{+}{+}$
7: Send $\mathcal{X}$ to client
Client:
8: Decrypt entries of $\mathcal{X}$ with k' and return them as $DB(w)$

Figure 5.4: Static searchable encryption PiBas [1].

one key for each instance and this requires from the client to store $O(\log N)$ keys. In order to reduce the local storage to $O(1)$, we can generate the key for each instance pseudorandomly from a single master secret key using a PRF. Instantiated with PiBas, SD_a requires a single roundtrip for retrieving the result $DB(w)$. Updates require one rountrip for retrieving the old indexes to be merged, and one more

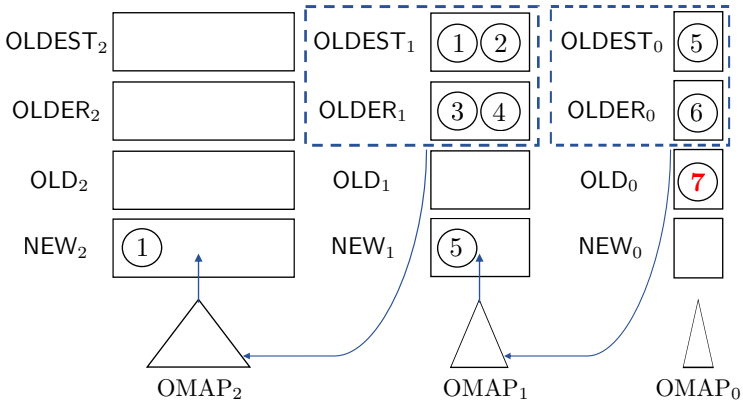


Figure 5.5: SD_d: from PiBas to DSE (de-amortized version). These are the encrypted indexes after 7 consecutive insertions $1-7$. Each level i contains 3 searchable encrypted indexes (OLDEST, OLDER, OLD) and one index (NEW) that is used for merging and rebuilding into a single index the OLDEST and OLDER indexes of the previous level $i-1$. The update algorithm passes through all levels and moves one element using OMAP_i from level $i-1$ to level i. Searching for a keyword w requires to search all the OLDEST, OLDER and OLD encrypted indexes and return to the client all the individual search results.

message from the client to the server (possibly "piggy-backed" to the next operation) for writing the new EDB_j.

With SD_a it is easy to clean-up deleted entries. During updates, before creating the merged EDB_j the client identifies all the entries in EDB_i, for $i < j$, that have corresponding deletions and removes them (padding with dummy records to fill up EDB_j).

5.2.2 De-amortized construction

Recall that our key goal is to design schemes with small client storage. SD_a has excellent performance, albeit in the amortized setting; during updates the client needs to download and locally rebuild an encrypted index. Most times, that index will be relatively small but once in a while this index will become very large (up

Let (KeyGen, Setup, Map, Search) refer to the PiBas routines [1], as described in Figure 5.4.

$\underline{(K, \sigma; EDB) \leftarrow \mathsf{Setup}(\lambda, N)}$

Set $\ell \leftarrow \lfloor \log N \rfloor$
for $i = 0, \ldots, \ell$ **do**
    Initialize OMAP_i with capacity 2^i
    Set OLDEST_i, OLDER_i, OLD_i and NEW_i to $\emptyset$
    Set $cnt_i \leftarrow 0$
Set $EDB \leftarrow \{\mathrm{OMAP}_i, \mathsf{OLDEST}_i, \mathsf{OLDER}_i, \mathsf{OLD}_i, \mathsf{NEW}_i, cnt_i\}_{i=0}^{\ell}$
Set $upd_{cnt} \leftarrow 0$
Set $\sigma \leftarrow \{upd_{cnt}, \mathsf{cnt}_i\}_{i=0}^{\ell}$ and the OMAP states
Set K an empty matrix of size $4 \cdot (\ell + 1)$
for $i = 0, \ldots, 3$ **do**
    for $j = 0, \ldots, \ell$ **do**
        $K[i][j] \leftarrow \mathsf{PiBas.KeyGen}(1^\lambda)$

$\underline{DB(w) \leftrightarrow \mathsf{Search}(K, q, \sigma; EDB)}$

Client $\leftrightarrow$ Server:
$\mathcal{X} \leftarrow \emptyset$.
for $i = \ell \cdots 0$ **do**
    if $\mathsf{OLDEST}_i \neq \emptyset$ **then**
        $\mathcal{X} \leftarrow \mathcal{X} \cup \mathsf{PiBas.Search}(K[i][0], \mathsf{OLDEST}_i)$
    if $\mathsf{OLDER}_i \neq \emptyset$ **then**
        $\mathcal{X} \leftarrow \mathcal{X} \cup \mathsf{PiBas.Search}(K[i][1], \mathsf{OLDER}_i)$
    if $\mathsf{OLD}_i \neq \emptyset$ **then**
        $\mathcal{X} \leftarrow \mathcal{X} \cup \mathsf{PiBas.Search}(K[i][2], \mathsf{OLD}_i)$

Client:
Decrypt the entries of $\mathcal{X}$ with K and parse them as (id, op)
$DB(w) \leftarrow \{id \mid (id, add) \in \mathcal{X} \wedge (id, del) \notin \mathcal{X}\}$

Figure 5.6: SD_d: from PiBas to DSE (de-amortized version).

to the entire DB) as it is shown in Figure 5.14(c), which invalidates our key goal. To overcome this obstacle, we present here a de-amortized version of our SD_a construction which we call SD_d. Unlike our amortized scheme that can work with any result-hiding static scheme, SD_d requires the setup process of the static scheme to be efficiently decomposable to discrete steps so that the necessary local state for executing each step is small and efficiently retrievable—PiBas is again a natural

Let (KeyGen, Setup, Map, Search) refer to the PiBas routines [1], as described in Figure 5.4.

$(K, \sigma; EDB) \leftrightarrow \mathsf{Update}(K, op, w, id, \sigma; EDB)$

Client $\leftrightarrow$ Server:

1: **for** $i = \ell, \ldots, 1$ **do**
2: **if** $\mathsf{OLDEST}_{i-1} \neq \emptyset \wedge \mathsf{OLDER}_{i-1} \neq \emptyset$ **then**
3: **if** $\mathsf{cnt}_i < 2^{i-1}$ **then**
4: Server sends to client $\mathsf{OLDEST}_{i-1}[cnt_i]$ who decrypts it with $K[i-1][0]$ and parses it as $(w', id,' op')$
5: **else** server sends to client $\mathsf{OLDER}_{i-1}[cnt_i \ \% \ 2^{i-1}]$ who decrypts it with $K[i-1][1]$ and parses it as (w', id', op')
6: $cnt_i \leftarrow cnt_i + 1$
7: Client computes num as the number of times NEW_i has been fully rebuilt and $c_w \leftarrow$OMAP$_i$.get(w', num)
8: **if** $c_{w'} = \perp$ **then** client sets $c_{w'} \leftarrow 0$
9: Client sets $c_{w'} \leftarrow c_{w'} + 1$ and runs OMAP$_i$.put$((w', num), c_{w'})$
10: Client sends to server $(key, value) \leftarrow$PiBas.Map$(w, id', op', c_{w'}, K[i][3])$
11: Server runs NEW_i.put$(key, value)$
12: **if** $|\mathsf{NEW}_i| = 2^i$ **then** ▷ Client can deduce this from upd_{cnt}
13: Server sets $\mathsf{OLDEST}_{i-1} \leftarrow \mathsf{OLD}_{i-1}$ and $\mathsf{OLDER}_{i-1} \leftarrow \emptyset$
14: **if** $\mathsf{OLDEST}_i = \emptyset$ **then** server sets $\mathsf{OLDEST}_i \leftarrow \mathsf{NEW}_i$ and client sets $K[i][0] \leftarrow K[i][3]$
15: **else if** $\mathsf{OLDER}_i = \emptyset$ **then** server sets $\mathsf{OLDER}_i \leftarrow \mathsf{NEW}_i$ and client sets $K[i][1] \leftarrow K[i][3]$
16: **else** server sets $\mathsf{OLD}_i \leftarrow \mathsf{NEW}_i$ and client sets $K[i][2] \leftarrow K[i][3]$
17: Client sets $K[i][3] \leftarrow$ PiBas.KeyGen(1^λ)
18: Client sets $K[0][3] \leftarrow$ PiBas.KeyGen(1^λ)
19: Client runs PiBas.Setup$(K[0][3], (w, id, op))$ and sends the output to server who stores it as NEW_0
20: **if** $\mathsf{OLDEST}_0 = \emptyset$ **then** server sets $\mathsf{OLDEST}_0 \leftarrow \mathsf{NEW}_0$ and client sets $K[0][0] \leftarrow K[0][3]$
21: **else if** $\mathsf{OLDER}_0 = \emptyset$ **then** server sets $\mathsf{OLDER}_0 \leftarrow \mathsf{NEW}_0$ and client sets $K[0][1] \leftarrow K[0][3]$
22: **else** server sets $\mathsf{OLD}_0 \leftarrow \mathsf{NEW}_0$ and client sets $K[0][2] \leftarrow K[0][3]$
23: Client sets $upd_{cnt} \leftarrow upd_{cnt} + 1$

Figure 5.7: SD_d: from PiBas to DSE (de-amortized version).

candidate, hence SD_d is specifically instantiated with it. The reason for this is that the key technical idea is inspired by the classic *lazy rebuild* technique of Overmars

and van Leeuwen [73]. The $O(2^i)$ steps necessary for running PiBas.Setup for a database of 2^i elements are split over the previous 2^i updates, executed one at a time.

With SD_d, four encrypted indexes OLDEST_i, OLDER_i, OLD_i, and NEW_i are maintained for each $i = 0, \ldots, \log N - 1$ (as it is shown in Figure 5.5). Each of the "old" indexes is either empty or contains exactly 2^i items. Moreover, if OLDEST_i is empty then so is OLDER_i, and if OLDER_i is empty then so is OLD_i. The fourth data structure NEW_i is either empty or a partially built index. The setup process of NEW_i is executed over 2^i updates. In this manner, we guarantee that each entry is stored in *exactly* one of the "old" encrypted index (across all sizes). Hence, the search protocol (Figure 5.6) is almost unchanged—the server just needs to search in at most three indexes per size.

Where SD_d strongly deviates from the amortized construction SD_a is during updates (Figure 5.7). Recall that we are using PiBas (Figure 5.4) but we are storing triplets of the form (w, id, del). The update algorithm passes through all sizes i from largest to smallest and moves one element from the set of indexes of level $i-1$ to the set of indexes of level i. For each level, if both OLDEST_{i-1}, OLDER_{i-1} are non-empty, this implies that within the next 2^i updates an index of size 2^i needs to have been fully rebuilt—else we would run out of space at level $i-1$! Therefore, one step of PiBas.Setup needs to be executed during each of these 2^i updates, moving one entry (w, id, op) from $\mathsf{OLDEST}_{i-1} \cup \mathsf{OLDER}_{i-1}$ into NEW_i. Moreover, to preserve forward privacy we must guarantee that this step does not reveal any information to the server. We explain how we achieve this next.

The EDB encrypted index of each PiBas instance contains one map T the keys and values of which are computed with the $\mathsf{PiBas.Map}$ function. For each of the 2^i updates, the client retrieves from the server and decrypts one entry from the maps T corresponding to OLDEST_{i-1} and OLDER_{i-1}, sequentially from beginning to end, i.e., treating the maps as arrays (the position of the next entry to retrieve can be computed efficiently based only on the current global update counter). The Map algorithm takes as input K, w, id, op, c where K is a key for PiBas freshly chosen every time the client starts rebuilding NEW_i, w, id, op are read from the retrieved entry, and c is a counter that counts how many times w has already appeared in the NEW_i index. Unfortunately, c cannot be stored locally in an efficient manner. In order to retrieve it, we deploy one oblivious map OMAP_i for each size 2^i that maps w to c. At every step, the client queries the corresponding OMAP_i, uses the retrieved c to run Map, increments it, and stores it back to the same OMAP_i.

This leaves one issue to be handled: Between rebuilds of NEW_i, the counters c need to be reset as PiBas searches always start from zero. Since the client cannot do that in one pass efficiently, we use an alternative approach. OMAP_i maps (w, num) to c, where num is the number of times NEW_i has previously been rebuilt (computable from)the current global update counter). When querying OMAP_i for (w, num), the client treats all returned entries with $num' < num$ as null and can safely overwrite them.

Every time NEW_i is fully built (i.e., has size 2^i), the server moves it to the oldest non-empty index among OLDEST_i, OLDER_i, OLD_i. Moreover, both OLDEST_{i-1} and OLDER_{i-1} are deleted since their purpose is served—all of their entries have been

moved to NEW_i. Then, if OLD_{i-1} exists, the server moves it to OLDEST_{i-1}. Finally, every update creates a "singleton" encrypted index at the oldest available slot for size 0 for the newly inserted entry. All PiBas instances are always instantiated with a freshly chosen key. Figure 5.5 illustrates the encrypted indexes after seven consecutive inserts.

Security. The backward privacy of SD_d is proven exactly in the same manner as that of SD_a since the search protocol is essentially the same. Forward privacy follows from these observations. First, for each update (w, id, op) the server sees a new PRF evaluation since we choose new PiBas keys for each instance and always increment the keyword counter for that keyword-instance combination. Second, our modified version of PiBas is response-hiding. Third, each update accesses a predetermined position in at most $2 \cdot \log N$ map data structures, and $\log N$ read/write oblivious map queries that do not reveal anything to the server about the accessed entries.

Theorem 5 *Assuming* PiBas *is an adaptively-secure result-hiding static SE scheme, and $OMAP_i$ are secure oblivious maps,* SD_a *is an adaptively-secure DSE according to Definition 2 with* $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = \perp$ *and* $\mathcal{L}_{\mathsf{SEARCH}}(w) = \boldsymbol{Updates}(w)$.

Proof 6 *(Sketch) Let* $\mathsf{Sim}_{\mathsf{PB}} = \{SimInit_{\mathsf{PB}}, SimSearch_{\mathsf{PB}}\}$ *be the simulator for* Pibas. *First, we observer that* $SimInit_{\mathsf{PB}}$ *can be decomposed into calls to a stateful* $SimInitOne_{\mathsf{PB}}$ *that simulates just one step of the setup simulation at a time. The input state of* $SimInitOne_{\mathsf{PB}}$ *is the partially built table, and the leakage N. After N executions,* $SimInitOne_{\mathsf{PB}}$ *provides an output that is identically distributed with that of* $SimInit_{\mathsf{PB}}$ *on input N. This follows easily by the fact that the setup process*

of Pibas *consists of populating a hash table with N semantically secure encryptions, stored at pseudorandomly computed positions. The simulator $SimInitOne_{\mathsf{PB}}$ just needs to remember its previous randomly chosen positions so that eventually he outputs the entire table.*[1]

With that observation, we build our simulator Sim *as follows. First, all calls to OMAP, are replaced by simulated accesses. During setup, SimInit launches $4\cdot(\ell+1)$ independent instances of $SimInitOne^{i}_{\mathsf{PB}}$ for $i = 0,\ldots,\ell$ and corresponding sizes $1,\ldots,2^{\ell}$, and initializes update counter $upd = 0$. For each update, whenever OLDEST_i, OLDER_i are full (which can be computed from i and upd), SimUpdate calls $SimInitOne^{i+1}_{\mathsf{PB}}$. If $SimInitOne^{i+1}_{\mathsf{PB}}$ is full (after $2^{i+1}+1$ calls), the simulator terminates the existing $SimInitOne^{i}_{\mathsf{PB}}$ instances mapped to OLDEST_i, OLDER_i and map the $SimInitOne^{i}_{\mathsf{PB}}$ instance of OLD_i to OLDEST_i (if it is not vacant). Moreover, it treats the $SimInitOne^{i+1}_{\mathsf{PB}}$ instance as mapped to the oldest vacant instance for size 2^{i+1}, and launches a new instance mapped to NEW_i. Finally, it always launches a new instance of $SimInitOne^{1}_{\mathsf{PB}}$, maps it to the oldest non-vacant instance for size 1, and increments upd. The search simulator SimSearch is identical to that of SD_a (it just has to call up to three instances of $SimSearch_{\mathsf{PB}}$ per size, depending on upd).*

By the same reasoning as that for SD_a above, and since $OMAP_i$ are independently instantiated with secure oblivious maps, the transcript produced by Sim *is indistinguishable from the messages observed by the adversary during the real*

[1]For simplicity, we assume that the first time $SimInitOne_{\mathsf{PB}}$ is called, it just simulates (internally) the key generation process, hence $SimInitOne_{\mathsf{PB}}$ will be called a total of $N+1$ times to emulate the execution of $SimInit_{\mathsf{PB}}$ on input N.

protocol execution. □

Efficiency. Updates require $O(\log N)$ OMAP queries. With the oblivious map of [41], their total access overhead is $O(\log^3 N)$, which is the dominating cost for updates. This is *worst-case* asymptotic update efficiency, as opposed to the amortized performance of SD_a. Search time is $O(a_w + \log N)$, same as that of the amortized version (up to three times slower due to multiple structures per size). Server storage is linear to the number of total updates; more concretely, the client chooses an upper bound on the total number of updates ahead of time and initializes the oblivious maps—for better server space efficiency, the above initialization can be split into multiple smaller steps, i.e., when the set of indexes of size $i-1$ becomes full we start initializing with dummy values the oblivious maps for the indexes of size $i+1$.

Crucially, permanent client storage is $\tilde{O}(\log^2 N)$ in order to store all the OMAP stashes and the PiBas keys. For appropriately chosen parameters, this is very small in practice since these stashes are usually sparsely populated (see experimental evaluation in Section 5.4). If we want to minimize client storage, there are two additional tricks: (i) stashes are stored at the server and downloaded as necessary during updates without affecting the update asymptotics, and (ii) the keys for PiBas are generated with a PRF from a single master secret key. These make the client storage $O(1)$. While describing SD_d in Figures 5.6 and 5.7, we assume parties store a counter cnt_i that is used to deduce which are the next elements to be retrieved for lazy rebuild. This is just for clarity of presentation; they can be efficiently computed by keeping a global update counter. Finally, although SD_d entails oblivious maps,

they are *only* used during updates and *not* during searches. As a result of this, while updates require $O(\log N)$ rounds of interaction, searches are still non-interactive.

Clean-up of is somewhat more involved with SD_d but it only affects the performance of updates and not searches. At a high level, the client maintains an additional OMAP OM_{del} accessed with key (w, id). During updates, while writing a record to NEW_i, for $i > 0$, the client looks up OM_{del}. If he receives $\perp$ he proceeds normally, else he writes a dummy value to NEW_i instead.

5.3 Efficient DSE with Quasi-Optimal Search

In this section, we present our third construction QOS that achieves quasi-optimal search time, according to Definition 5. The only existing backward-private constructions that achieve this are Orion and Horus from [66]. Both these schemes replace each of the n_w accesses necessary for retrieving the result $DB(w)$ with an oblivious map/oblivious RAM access. Contrary to this, QOS requires a single read and write to an oblivious map during search (independently of n_w); the remaining computation for retrieving the result is executed at the server by traversing a tree data structure that serves as a "pivot" to identify deleted entries.

The basic idea behind QOS is described in Figure 5.8. Consider a full binary tree with N leafs, where N is an upper bound on the total number of insertions in the DSE (N can also serve as a trivial bound for the number of deletions). The function $label(v)$ returns a value in $[1, 2N-1]$ which is the result of the "natural" labeling of tree nodes as follows: The N leafs are labeled from leftmost to rightmost

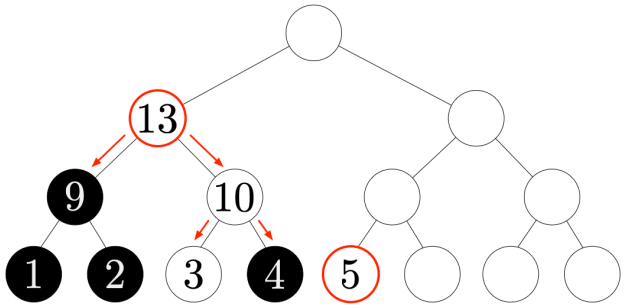


Figure 5.8: Update tree for QOS with maximum insertions $N = 8$. Nodes are labeled with $[1, 15]$ leafs-to-root and left-to-right. This is the tree state after five insertions 1-5, and three deletions for $1, 2, 4$. A subsequent search starts from the Best Range Cover of leafs $[1, 5] = (13, 5)$ and proceeds downwards until it finds a black node or a leaf. The result is $(3, 5)$.

with $1, \ldots, N$. The remaining nodes are labeled in an increasing order per level and from left to right, e.g., the parent of the two leftmost leafs is labeled with $N + 1$, its right sibling with $N + 2$, and so on, all the way to the root that is labeled with $2N - 1$. Every node has a corresponding color $c_v \in \{\text{white,black}\}$; all nodes are initially white. The client holds a "conceptual" tree like this for every keyword w. In said tree, inserted entries correspond to leafs that are being populated from *left to right*, i.e., after i_w insertions (and without deletions), the result of the search is related to the i_w first leafs with labels $1, \ldots, i_w$.

For each deletion, the client needs to mark one node as black. Assuming the deleted entry was previously stored at the j-th leaf, this is the node that will be marked as black. However, additional nodes may be marked black according to the following simple rule: "if both children of a node are black, it is also marked black." Hence, for the above deletion the client needs to access the colors of all the ancestors of the j-th leaf and their siblings. With this information, he can update their colors accordingly. Simply, each deletion "eliminates" an entire subtree by marking its

F is a PRF, $RND = (Gen, Enc, Dec)$ is a semantically secure symmetric encryption scheme, and H, H' are hash functions.

$(K, \sigma; EDB) \leftarrow \mathsf{Setup}(1^\lambda)$

1: Initialize OMAPs OM_{del}, OM_{state} of capacity N
2: Initialize OMAP OM_{cnt} of capacity $|W|$
3: Initialize empty maps $\mathcal{D}, \mathcal{I}$
4: Set $EDB \leftarrow \{OM_{cnt}, OM_{del}, OM_{state}, \mathcal{D}, \mathcal{I}\}$
5: $k_{\mathcal{I}} \leftarrow F.Gen(1^\lambda)$, $k_{\mathcal{D}} \leftarrow F.Gen(1^\lambda)$,
$k \leftarrow RND.Gen(1^\lambda)$
6: State σ contains the OMAP states
7: Key K contains $k_{\mathcal{I}}, k_{\mathcal{D}}, k$

Figure 5.9: QOS: DSE with quasi-optimal search time $O(n_w \log(i_w))$.

root black.

During a search after i_w insertions, the leafs that contain the result can be reached as follows. First, we compute the Best Range Cover for leafs with labels $[1, i_w]$. Then, starting independently from each node in the Best Range Cover the search progresses downwards towards the leafs. If it encounters a black node it stops (knowing that there is no undeleted entry below). Upon reaching a leaf that is not black, the corresponding entry is added to the result. In our analysis we show that, while the entire subtree that covers the leafs $[1, i_w]$ is of size $< 2i_w$, the nodes that are accessed during this process are $O(n_w \log i_w)$. Next, we describe our scheme in detail and we explain the implementation decisions we made in order to hide the necessary actions for manipulating this tree.

Setup. During Setup (Figure 5.9), the client initializes three empty OMAPs with capacity $|W|$, N, N, respectively:

(i) OM_{cnt} maps keywords w to cnt_w and i_w, where cnt_w is the number of previous

F is a PRF, $RND = (Gen, Enc, Dec)$ is a semantically secure symmetric encryption scheme, and H, H' are hash functions.

$\underline{DB(w) \leftrightarrow \mathsf{Search}(K, q, \sigma; EDB)}$

Client:

1: $(cnt_w, i_w) \leftarrow OM_{cnt}.\text{get}(w)$
2: $tk_{\mathcal{I}} \leftarrow F(k_{\mathcal{I}}, (w, cnt_w))$; $tk_{\mathcal{D}} \leftarrow F(k_{\mathcal{D}}, (w, cnt_w))$
3: $cnt_w \leftarrow cnt_w + 1$; $OM_{cnt}.\text{put}(w, (cnt_w, i_w))$
4: Send $(tk_{\mathcal{I}}, tk_{\mathcal{D}}, i_w)$ to server

Server:

5: $d_0, \ldots, d_m \leftarrow$ labels of Best Range Cover for leafs $[1, i_w]$
6: $(\mathcal{X}, \mathcal{Y}) \leftarrow (\emptyset, \emptyset)$ ▷ $\mathcal{X}$ will contain encrypted result, $\mathcal{Y}$ the labels of black nodes encountered in search
7: **for** $i = 0 \ldots m$ **do**
8: $(\mathcal{X}, \mathcal{Y}) \leftarrow (\mathcal{X}, \mathcal{Y}) \cup \mathsf{RecSrc}(EDB, tk_{\mathcal{I}}, tk_{\mathcal{D}}, d_i)$
9: Send $\mathcal{X}, \mathcal{Y}$ to client

Client:

10: $(DB(w), \mathcal{X}', \mathcal{Y}') \leftarrow (\emptyset, \emptyset, \emptyset)$
11: **for** $x \in \mathcal{X}$ **do**
12: $(id, leaf) \leftarrow RND.Dec(k, x)$
13: $tk \leftarrow F(k_{\mathcal{I}}, (w, cnt_w))$; $key \leftarrow H(tk, (w, leaf))$
14: $value \leftarrow RND.Enc(k, (id, leaf))$
15: $\mathcal{X}' \leftarrow \mathcal{X}' \cup (key, value)$
16: $DB(w) \leftarrow DB(w) \cup id$
17: **for** $y \in \mathcal{Y}$ **do**
18: $tk \leftarrow F(k_{\mathcal{I}}, (w, cnt_w))$, $key \leftarrow H'(tk, (w, y))$
19: $\mathcal{Y}' \leftarrow \mathcal{Y}' \cup (key, 1)$
20: Shuffle each of $\mathcal{X}'$, $\mathcal{Y}'$ and send them to server

Server:

21: **for** $(key, value) \in \mathcal{X}'$ **do** $\mathcal{I}.\text{put}(key, value)$
22: **for** $(key, value) \in \mathcal{Y}'$ **do** $\mathcal{D}.\text{put}(key, value)$

$\underline{(\mathcal{X}, \mathcal{Y}) \leftarrow \mathsf{RecSrc}(EDB, tk_{\mathcal{I}}, tk_{\mathcal{D}}, d)}$

1: **if** $\mathcal{D}.\text{get}(H'(tk_{\mathcal{D}}, d)) = 1$ **then return** $(\emptyset, d)$
2: **if** d is a leaf **then return** $(\mathcal{I}.\text{get}(H(tk_{\mathcal{I}}, d), \emptyset)$
3: Let d_l, d_r be the labels of the left and right child of d
4: $(\mathcal{X}_l, \mathcal{Y}_l) \leftarrow \mathsf{RecSrc}(EDB, tk_{\mathcal{I}}, tk_{\mathcal{D}}, d_l)$
5: $(\mathcal{X}_r, \mathcal{Y}_r) \leftarrow \mathsf{RecSrc}(EDB, tk_{\mathcal{I}}, tk_{\mathcal{D}}, d_r)$
6: **return** $(\mathcal{X}_l \cup \mathcal{X}_r, \mathcal{Y}_l \cup \mathcal{Y}_r)$

Figure 5.10: QOS: DSE with quasi-optimal search time $O(n_w \log(i_w))$.

F is a PRF, $RND = (Gen, Enc, Dec)$ is a semantically secure symmetric encryption scheme, and H, H' are hash functions.
$(K, \sigma, EDB) \leftrightarrow \mathsf{Update}(K, op, w, id, \sigma; EDB)$

Client:

1: $(cnt_w, i_w) \leftarrow OM_{cnt}.\text{get}(w)$
2: **if** $op = add$ **then**
3: $\quad OM_{cnt}.\text{put}(w, (cnt_w, i_w + 1))$
4: $\quad OM_{del}.\text{put}((w, id), i_w + 1))$
5: $\quad tk \leftarrow F(k_{\mathcal{I}}, (w, cnt_w))$
6: $\quad key \leftarrow H(tk, (w, i_w + 1)),$
$\quad value \leftarrow Enc(k, (id, i_w + 1))$
7: **else** $\quad \triangleright\ op = del$
8: $\quad pos \leftarrow OM_{del}.\text{get}(w, id)$
9: $\quad d_0 \dots d_p \leftarrow$ labels of ancestors of pos $\quad \triangleright\ d_0 = pos$
10: $\quad d'_0 \dots d'_p \leftarrow$ labels of siblings of $d_0 \dots d_p \triangleright\ d'_p = \perp$
11: $\quad$ **for** each d_i **do** color $c_i \leftarrow OM_{state}.\text{get}(w, d_i)$
12: $\quad$ **for** each d'_i **do** color $c'_i \leftarrow OM_{state}.\text{get}(w, d'_i)$
13: $\quad c_0^{new}, \dots, c_p^{new} \leftarrow$ Update the colors c_i
14: $\quad$ Let $j \leftarrow \max\{\ i \mid c_i^{new} \neq c_i \wedge c_i^{new} = \text{black}\}$
15: $\quad OM_{state}.\text{put}((w, d_j), c_j^{new})$
16: $\quad tk \leftarrow F(k_{\mathcal{D}}, (w, cnt_w))$
17: $\quad key \leftarrow H'(tk, (w, d_j));\ value \leftarrow 1$
18: Send $(key, value)$ to server

Server:

19: **if** $op = add$ **then** $\mathcal{I}.\text{put}(key, value)$
20: **else** $\mathcal{D}.\text{put}(key, value)$

Figure 5.11: QOS: DSE with quasi-optimal search time $O(n_w \log(i_w))$.

searches for w, and i_w is number of previous insertions for w.

(ii) OM_{del} maps each keyword-file identifier pair w, id to $label(v)$ where v is the leaf to which it was inserted; during deletions, this is used to retrieve the "position" of the entry to be deleted.

(iii) OM_{state} maps a keyword-node label pair $w, label(v)$ to the color of the node v.

The encrypted index EDB stored at the server consists of the oblivious maps and

two empty maps $\mathcal{I}, \mathcal{D}$ of capacity N, D respectively (D is an upper bound on deletions that can also serve as the capacity of OM_{state}; trivially it can be set to $O(N)$). The client stores locally the states of the three oblivious maps, two PRF keys $k_{\mathcal{I}}, k_{\mathcal{D}}$ and a symmetric encryption key k.

Update. For updates (Figure 5.11), the client first retrieves the number of previous searches cnt_w and the insertion count i_w via OM_{cnt}. Then, we describe the two cases separately. For insertions (lines 2-6), the client increments the update count and writes it to OM_{cnt}. He also writes an entry at OM_{del} that maps (w, id) to the leaf location where it is stored in $\mathcal{I}$ (this will be used for deleting this entry in the future). Finally, the client encrypts $id, i_w + 1$. The resulting ciphertext is stored at the server in map $\mathcal{I}$ at a location computed by the hash function H, using a token tk that the client computes pseudorandomly with $k_{\mathcal{I}}$ for $(w, i_w + 1)$.

For deletions (lines 7-17), the client retrieves the label *pos* of the tree leaf at which w, id has been stored via OM_{del}. Then, he computes the labels of all the ancestors and the siblings of the ancestors of *pos*, and retrieves their colors from OM_{state} (lines 9-12). With these, he can update the colors of all the ancestors of *pos* (in the simplest case, *pos* is set to black, more generally this deletion may cause some of its ancestors to become black too). Finally, the client finds d_j, the furthest ancestor of *pos* that was first set to black during this deletion. He then marks an entry at $\mathcal{D}$ (at the server) at a location computed by the hash function H', using a token tk that the client computes pseudorandomly with $k_{\mathcal{D}}$ for (w, d_j), as well as store its new color at OM_{state}.

Search. During searches Figure 5.10, the client first retrieves cnt_w, i_w from OM_{cnt} and pseudorandomly computes two search tokens for w, cnt_w: (i) $tk_{\mathcal{I}}$ is computed with $k_{\mathcal{I}}$ and will be used to retrieve the result, and (ii) $tk_{\mathcal{D}}$ is computed with $k_{\mathcal{D}}$ and will be used to identify black nodes encountered during the search, corresponding to deletions. These tokens and i_w are sent to the server. The client also increments the search counter cnt_w and stores it to OM_{cnt}.

The server first computes the set of tree nodes $d_0, \ldots, d_m$ that constitute the Best Range Cover of leaf nodes $[0, i_w]$—each entry of $DB(w)$ will be related to a descendant of one of d_i. The search process is quite simple and it entails a recursive search process starting from each of d_i and progressing downwards at the tree (Figure 5.9, Algorithm RecSrc). At each node d, the server checks whether the location $H'(tk_{\mathcal{D}}, d)$ has been written at $\mathcal{D}$, in which case, this is a "black" node, i.e., any previously inserted entries at the subtree with root d have since been deleted. Hence, the server can simply record its node label and return. Otherwise, he proceeds to parse its children. Upon reaching a leaf that is not black, the server returns the encrypted entry from $\mathcal{I}$ at position $H(tk_{\mathcal{I}}, d)$—since d is a non-deleted leaf, it corresponds to an entry of $DB(w)$.

The server returns to the client all retrieved values from $\mathcal{I}$ and all marked entries from $\mathcal{D}$ that correspond to black nodes encountered during the tree traversal (and removes them from $\mathcal{I}, \mathcal{D}$). The client computes $DB(w)$ by decrypting the first ones. Finally, he "re-maps" all the entries of $\mathcal{I}$ and $\mathcal{D}$, using new pseudorandom tokens with keys $k_{\mathcal{I}}, k_D$ respectively but increased search counter cnt_w, and sends them back to the server who stores them at $\mathcal{I}$ and $\mathcal{D}$.

Security. QOS is forward-private because during updates the server observes two types of accesses: (i) a fixed number of oblivious map operations (depending on the type of update) that reveal nothing, and (ii) a pair $(key, value)$ that consists of the outputs of a hash function modeled as a random oracle, and a semantically secure ciphertext. The latter clearly reveals nothing. For the former, note that we ensure that the same input is never passed to the random oracle twice during updates. This follows from incrementing i_w during insertions and from the fact that deletions never mark the same node as black. Since the input to the random oracle contains a token computed from a PRF for which the server does not have the key (and is only revealed during a future search), querying the oracle for "valid" values not previously seen is infeasible. Finally, note that after every search both tokens are changed so the server cannot connect future updates with ones prior to the search.

Regarding backward privacy, during searches the server learns the PRF tokens $k_{\mathcal{I}}$, $k_{\mathcal{D}}$ which allows him to compute the $\mathcal{I}, \mathcal{D}$ locations that he needs to access. This also allows him to recall when these entries in $\mathcal{I}, \mathcal{D}$ where made, i.e., the timestamp and type for all update operations for the queried keyword w. Moreover, since the topology of the tree is revealed to the server and the leafs of the tree are naturally mapped to timestamps of insertions, the server can deduce exactly which deletion canceled which prior insertion. As a result of this, QOS achieves BP-III.

Theorem 6 *Assuming F is a PRF, RND is a semantically secure encryption scheme, and the three OMAPs are secure oblivious maps,* QOS *is an adaptively-secure DSE according to Definition 2 in the programmable random oracle model,*

with $\mathcal{L}_{\mathsf{UPDATE}}(op, w, id) = op$ *and* $\mathcal{L}_{\mathsf{SEARCH}}(w) = (\boldsymbol{Updates}(w), \boldsymbol{DelHist}(w))$.

Proof 7 *We prove the security of* QOS *by defining a sequence of games as follows:*

- ***Game***-0***:*** *This is the* Real^{SSE} *game as defined in Section 5.1.*
- ***Game***-1***:*** *This is the same as Game*-0 *but during setup the OMAP initializations are replaced with calls to the OMAP simulators for sizes* W, N, N *respectively. All future OMAP accesses are emulated by calls to the corresponding access simulators. Game*-1 *is indistinguishable from Game*-0 *due to the security of the oblivious maps.*
- ***Game***-2***:*** *This is the same as Game*-1*, except that the encryptions value computed during update and search are all replaced with dummy zero encryptions. Game*-2 *is indistinguishable from Game*-1 *due to the semantic security of RND.*
- ***Game***-3***:*** *This is the same as Game*-2*, except that the tokens* $tk_{\mathcal{I}}, tk_{\mathcal{D}}$ *generated during update and search are generated uniformly at random from the range of the PRF* F*,* $\{0,1\}^{\lambda}$*. The first time a token is created for a certain* w, cnt_w *combination it is appended to one of the two lists* $\mathsf{TokensI}(w)$*,* $\mathsf{TokensD}(w)$ *(for insertions and deletions respectively), that are different for every keyword. Game*-3 *is indistinguishable from Game*-2 *due to the security of the PRF.*
- ***Game***-4***:*** *This is the same as Game*-3*, except that calls to* H *are replaced with a programmable random oracle as follows. For general* H*-calls from the adversary, if the input has not be queried before and the result has not been programmed, return a value chosen uniformly at random from the range of*

H and store the input-result pairs for future consistency. Else, return the previously stored result for this input.

During insertion updates (line 6), $H-$ calls are entirely eliminated and instead key is chosen uniformly at random from the range of H. The client holds a list T_I where he appends the chosen key. If the update is a deletion he appends $\perp$. Note that this also eliminates token generation at line 5.

Then, during search, let $U = (u_1, op_1), \ldots, (u_{a_w}, op_{a_w})$ be the list of timestamp-update type pairs corresponding to all previous updates for the queried keyword w, sorted by timestamp in increasing order. Let $u'_1, \ldots, u'_{i_w}$ be the sub-list of U such that $op_i = add$, again sorted in increasing order, and let $d_1, \ldots, d_{i_w}$ be the natural ordering of u'_i from $1 \ldots, i_w$. The client then programs the oracle such that $H(tk_{\mathcal{I}}, d_i) = T_I[u'_i]$. If $H(tk_{\mathcal{I}}, d_i)$ has been set previously (due to an adversarial query involving $tk_{\mathcal{I}}$ before this token was revealed), then the game aborts. Finally, line 13 of the search algorithm is replaced with choosing key uniformly at random from the range of H. Let $d_j = leaf$, then client sets $T_I[u'_j] = key$, in preparation of future searches.

First, note that unless the game aborts it produces a transcript identical to Game-3, in the programmable random oracle model for H. Given that the range of H is $\{0,1\}^\lambda$, whereas the total number of H-calls that the adversary can do beyond the ones required during searches is polynomial in λ (since the adversary is PPT), the probability of aborting is negligible in λ, hence Game-4 is indistinguishable from Game-3.

- ***Game*-5*:*** *This is the same as Game-4 but we now also replace H' with*

a programmable random oracle. For general H'-calls from the adversary, if the input has not been queried before and the result has not been programmed, return a value chosen uniformly at random form the range of H and store the input-result pairs for future consistency. Else, return the previously stored result for this input.

During deletion updates (line 17), $H'-$ calls are entirely eliminated and instead key is chosen uniformly at random from the range of H'. The client holds a list T_D where he appends the chosen key. If the update is an insertion he appends $\perp$. Note that this also eliminates token generation at line 16.

Then, during search, let $Dels = (v_1, v'_1), \ldots, (v_{d_w}, v'_{d_w})$ be the list of all timestamp-pairs that match each deletion timestamp v_i to the timestamp v'_i of the previous insertion it cancels out, sorted in increasing order such that $v_i > v_{i-1}$. Using U (from Game-4) and $Dels$ the client builds the entire update tree for w as follows. First create an empty binary tree with $2^{\lceil \log i_w \rceil}$ leafs. Match each leaf $[1, i_w]$ to an insertion operation's timestamp u'_i (as computed in Game-4) starting from the leftmost leaf. Then, for every $v_i \in Dels$, mark the leaf with timestamp v'_i as black and then keep moving upwards, reading at every level its ancestor and the sibling of its ancestor. If both children of a node is black mark it black. After finishing all steps for deletion with timestamp v_i, let d'_i be the node closest to the root that you just marked black. The client then programs the oracle such that $H'(tk_{\mathcal{D}}, d'_i) = T_D[v_i]$. If $H'(tk_{\mathcal{D}}, d'_i)$ has been set previously (due to an adversarial query involving $tk_{\mathcal{D}}$ before this token was revealed), then the game aborts.

Finally, line 18 of the search algorithm is replaced with choosing key uniformly at random from the range of H'. Let $d'_j = y$, then client sets $T_I[v_j] = key$, in preparation of future searches.

First, note that unless the game aborts it produces a transcript identical to Game-3, in the programmable random oracle model for H. This holds since the combination of $U, Dels$ uniquely define the colors of the nodes of the update tree for w. Then, using the same argument as above but for H', we conclude that Game-5 is indistinguishable from Game-4.

- ***Game*-6** : *This is the same as Game-5 but client receives op instead of op, w, id during updates, and $\boldsymbol{Updates}(w), \boldsymbol{DelHist}(w)$ instead of w during searches. Since he does not have access to w, he populates lists* $\mathsf{TokensI}$, $\mathsf{TokensD}$ *as follows. For a search at timestamp $\hat{w}$, the client first checks whether the input update history $\boldsymbol{Updates}(w)$ is an extension of one observed during a previous search that took place during timestamp $\hat{w}'$. If so, this implies that the searches at times $\hat{w}$ and $\hat{w}'$ are for the same keyword and he retrieves $tk_{\mathcal{I}}, tk_{\mathcal{D}}$ as the latest entries from $\mathsf{TokensI}(\hat{w}')$, $\mathsf{TokensD}(\hat{w}')$. Else, he chooses fresh random tokens $tk_{\mathcal{I}}, tk_{\mathcal{D}}$ and appends them to $\mathsf{TokensI}(\hat{w})$, $\mathsf{TokensD}(\hat{w})$.*

  *The client's code as described in **Game**-6, is essentially the code of the simulator in the Ideal^{SSE} game since it only takes as input the leakage specified in Theorem 6. By a standard hybrid argument, the produced transcript is indistinguishable from the one produced in **Game**-0, and the result follows.*

□

Efficiency. Updates with QOS require $O(\log N)$ OMAP queries resulting to a total of $O(\log^3 N)$ operations and $\log N$ roundtrips, using the OMAP of [41]. Setup is linear to N, D, the upper bound on insertions and deletions, as is the server's storage. The search time can be computed as follows. The OMAP queries take $O(\log^2 |W|)$ operations. Computing the Best Range Cover takes $O(\log i_w)$, same as the cover size itself. Parsing the tree in order to retrieve the result, takes $O(n_w \log i_w)$ since n_w leafs will be reached and the maximum height from each of them to the one of the nodes in the Best Range Cover is $\log i_w$. Even if every node along this traversal has a black sibling (which is a huge overestimation), the total number of black nodes encountered is $O(n_w \log i_w)$ as well. From all the above, the total search overhead with QOS is $O(n_w \log i_w + \log^2 |W|)$ and it takes $O(\log |W|)$ rounds of interaction.

The client's permanent storage is $O(\log^2 N)$ due to the OMAP stashes. If necessary, this can again be reduced to $O(1)$ at no asymptotic cost by storing the stashes at the server.

5.4 Experimental Evaluation

We implemented our three schemes in C++ in order to benchmark their performance and compare them with previous works. We used the OpenSSL [74] library for AES for our PRF and semantically secure encryption. For our experiments we used r5.8xlarge AWS machines with 32-core Intel Xeon 8259CL 2.5GHz processor, running Ubuntu 16.04 LTS, with 256GB RAM, 100GB SSD (GP2), and AES-NI enabled. All schemes were instantiated on a single machine with in-memory storage.

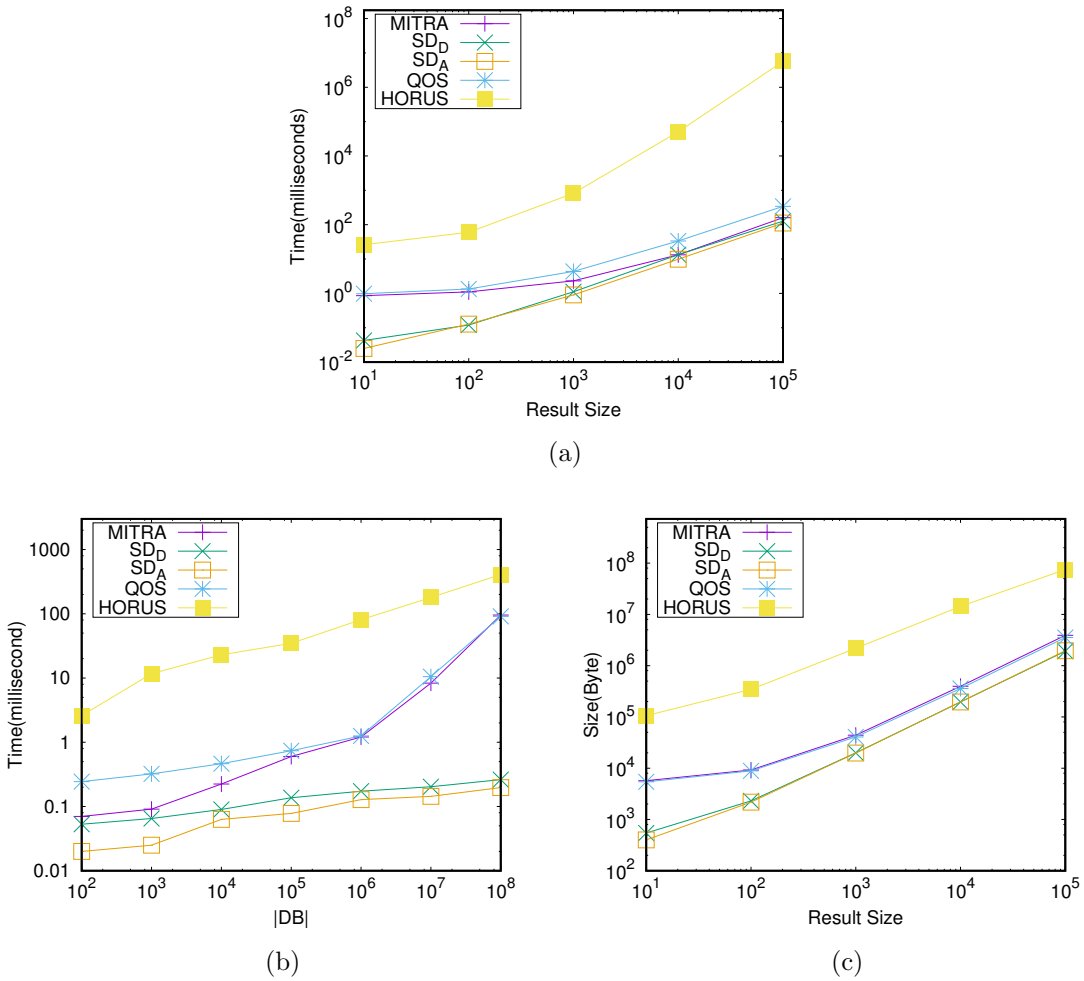


Figure 5.12: Synthetic Dataset—Search (a) computation time vs. variable result size for $|DB| = 1$M, (b) computation time vs. variable $|DB|$ for result size 100, (c) communication size vs. variable result size for $|DB| = 1$M.

We compared SD_a and SD_d with the previous state-of-the-art schemes with small client storage which can be achieved by the "word counter + oblivious map" approach. As described in the introduction, several schemes can be used in this manner, but Mitra [66] is simultaneously the most efficient and most secure (BP-II). For QOS, the main competitor is Horus [66] which is the fastest existing quasi-optimal scheme. Orion achieves BP-I but it is considerably slower in practice. For both these schemes, we used the code provided in [75].

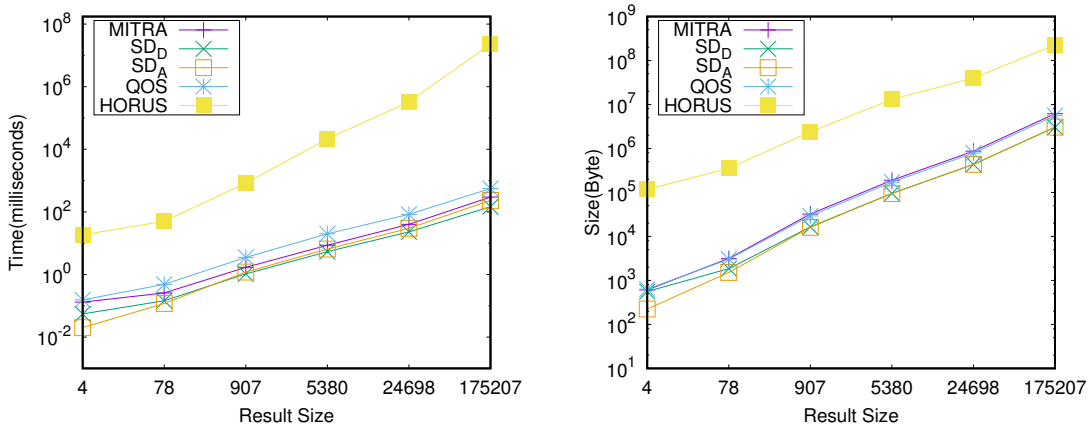


Figure 5.13: Crime Dataset—Search (a) computation time vs. variable result size, (b) communication size vs. variable result size.

Since we do not adopt a "clean-up" phase for SD_a and SD_d, for fairness we also run MITRA without clean-up (this is faster than the MITRA* numbers reported in [66] by up to 50%). We stress that both schemes are compatible with clean-up, with an additional update cost for SD_d and at no additional search cost for either one. For SD_a and SD_d we used one additional optimization, by storing the first 10 levels of the index collections locally. As we demonstrate below, the effect of this on local storage is small enough to be negligible, but it helps further improve their performance otherwise.

Our basic efficiency measurement is computation time and total communication size for search and update operations. In our experimental evaluation, we consider as unit-size a record/tuple (e.g., x-axis in Figures 5.12(b), 5.14(a) and 5.14(b))—the database in bytes can be obtained by multiplying the number of tuples with the tuple size (e.g., in crime dataset [46] the tuple-size is 210 bytes). For real-world application a tuple size may vary from a few bytes to GB, but in this

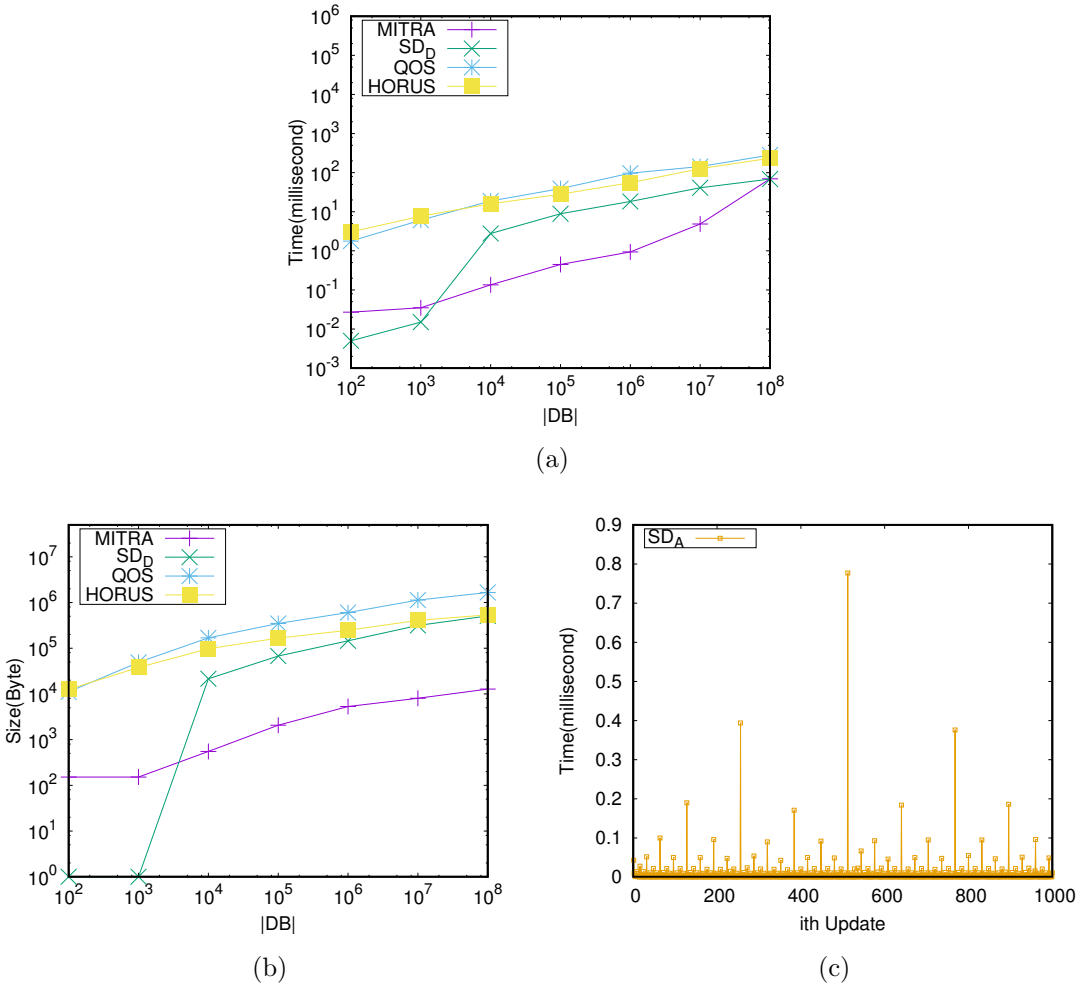


Figure 5.14: Synthetic Dataset—Update (a) computation time vs. variable $|DB|$, (b) communication size vs. variable $|DB|$, (c) computation time with SD_a for 1000 updates staring from empty DB.

section we focus only on the index costs/overheads, i.e., find the tuple/record identifiers that satisfy the encrypted queries, ignoring the costs for locating-downloading-decrypting the actual tuples/records, since the latter cost is common for all DSE schemes.

We consider variable datasets of synthetic records and size $|DB| = 10^2$–10^8 records, setting $|W|$ (i.e., the total number of keywords) to one-hundredth of $|DB|$. We also vary the search result size between 10–10^5. Each record, i.e., keyword-

document id pair, of our synthetic datasets consists of a 4-byte integer index file and an alphanumeric keyword of size ≤ 11 characters. We create the dataset to contain keywords with variable result sizes between 10–10^5 records. For instance, for a tested result size x, we create a random keyword with x random files identifiers and we distribute uniformly at random the remaining records to the remaining keywords. We also repeated the experiments on a real dataset (see Figures 5.13 and 5.16) consisting of 22 attributes and $6,123,276$ records of reported crime incidents in Chicago [46]. The used query attribute is the *location description* which contains 170 distinct keywords. Among these keywords the one with minimum frequency contains 1 record, while the one with maximum frequency comprises $1,631,721$ records.

In our experiments, before searching for w we delete at random 10% of its corresponding entries (unless stated otherwise), in order to show the impact of deletions in the search performance. The average of 10 executions is reported.

5.4.1 Search performance

Computation time. Figure 5.12(a) shows the execution time when searching for different result sizes and Figure 5.12(b) for different database sizes. First, we note that the time increases more steeply with larger result sizes than with larger $|DB|$, as expected. Second, for small result sizes SD_a and SD_d are much faster than Mitra. E.g., for $|DB(w)| = 10$ they are 85× and 20× faster than Mitra, respectively. This comes naturally as, for such sizes, the OMAP overhead of Mitra is dominating.

Concretely, for retrieving 100 result records from a dataset of size 10^6, SD_a takes 0.09ms, SD_d 0.12ms, and MITRA 1.11ms. As $|DB(w)|$ grows, the OMAP overhead becomes less important, and the performance of the three schemes converges, e.g., for 10^4 and $|DB| = 10^6$, SD_a takes 9.8ms, SD_d 13.3ms, and MITRA 13.2ms.

QOS has tremendously better search performance compared to the previous best quasi-optimal scheme HORUS, ranging from 4.4 up to 16531× faster. This comes naturally as the number of oblivious operations for HORUS is $O(|DB(w)|)$ ORAM accesses, whereas for QOS it is a single OMAP access to retrieve the counter a_w. E.g., for $|DB(w)| = 10^4$ and $|DB| = 10^6$, HORUS takes approximately 50sec and QOS takes 33.5ms. The performance of QOS is worse than MITRA, which is explained from the relatively small deletion rate (10%)—quasi-optimal schemes like QOS perform better for large deletion rates (for 10% deletions a_w is very close to n_w).

Communication size. Figure 5.12(c) shows the search communication size when $|DB(w)|$ varies between 10-10^5 for $|DB| = 10^6$. For all schemes, communication is increasing almost linearly with the result size. One exception is QOS and MITRA where for small result sizes (e.g., < 1000) because the communication cost is dominated by the OMAP operations. In practice, QOS requires 19-53× less communication than HORUS whose overhead is dominated by the ORAM accesses. For $|DB(w)| = 1000$, QOS sends 40KB whereas HORUS sends 2MB. Furthermore, SD_d requires 2-10× and SD_a 14× smaller communication size for search than MITRA, since both in SD_a and SD_d search does not depend on oblivious accesses. For the

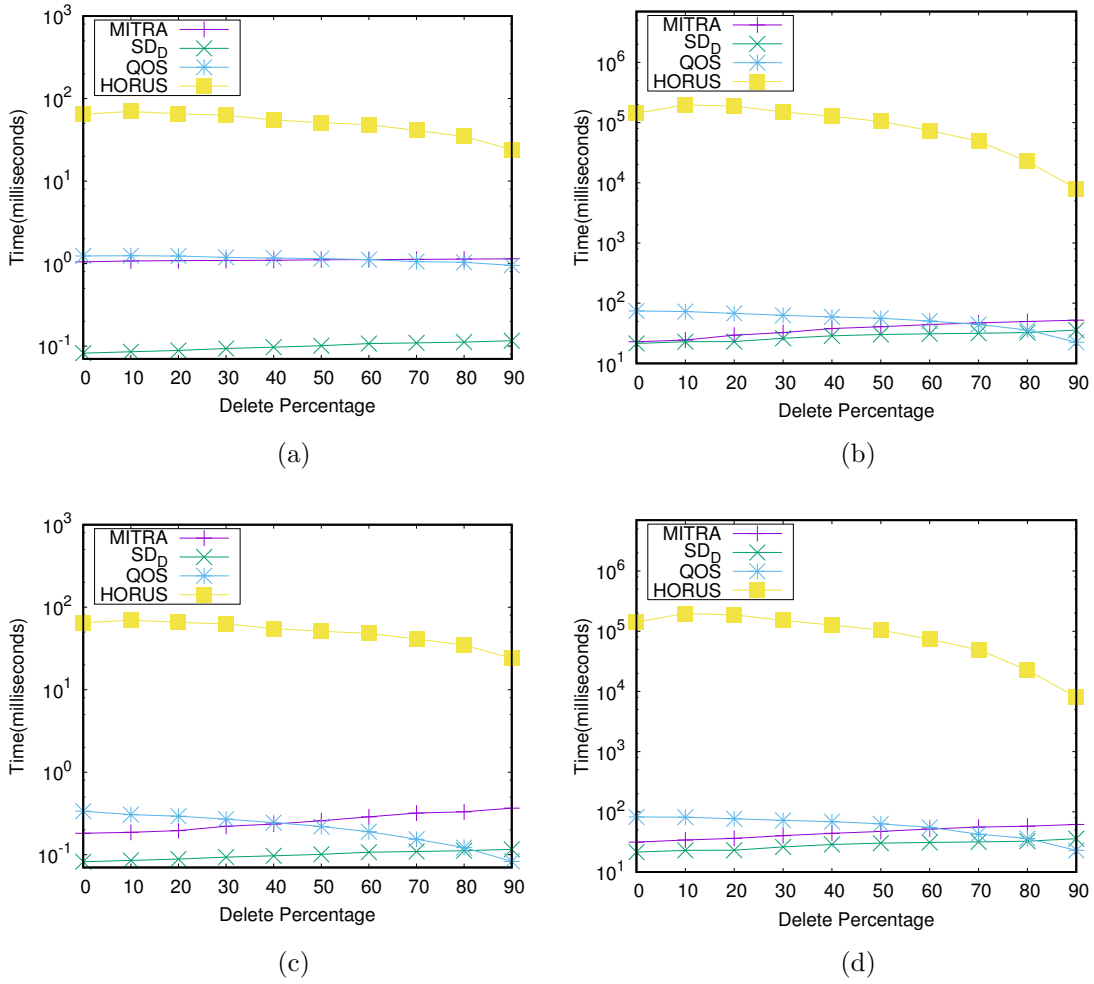


Figure 5.15: Synthetic Dataset—Search computation time for $|DB| = 1$M and variable deletion percentage for: (a) $i_w = 100$ using OMAP, (b) $i_w = 20$K using OMAP, (c) $i_w = 100$ storing word counters locally, (d) $i_w = 20$K storing word counters locally.

same result size as above, SD_a sends 19.7KB, SD_d 19.9KB, and Mitra $44KB$.

In addition to the results presented in Figure 5.12, we also compared our schemes with Mitra with local storage (which can be up to $O(N)$ as we have discussed in the Introduction). We measured the search computation time and communication size of variable database size between $10^2 - 10^6$ and we report that Mitra is more efficient than QOS, has similar performance with SD_d, while it is

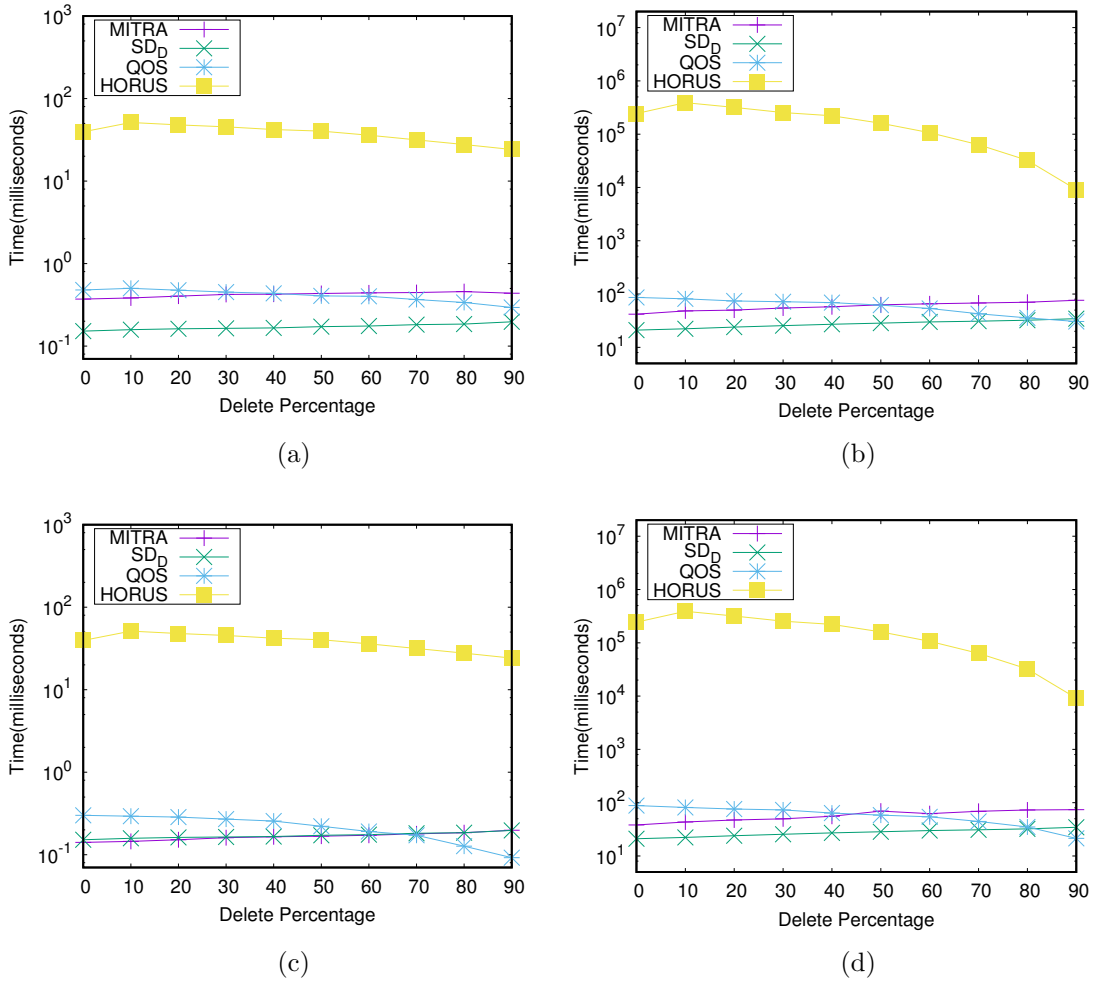


Figure 5.16: Crime dataset—Search computation time and variable deletion percentage for: (a) $i_w = 78$ using OMAP, (b) $i_w = 24698$ using OMAP, (c) $i_w = 78$ storing word counters locally, (d) $i_w = 24698$ storing word counters locally.

worse than SD_a. In Figure 5.13, we repeated the above experiments for the search costs, i.e., communication time and computation size, for the crime dataset and we observe similar conclusions.

5.4.2 Update performance

Computation time. Figure 5.14 shows (a) the update computation time and (b) the update communication size for variable database sizes for all schemes except for

SD_a, which has *amortized* update cost. The update performance of SD_a is reported in Figure 5.14(c), which shows the update time (step-by-step) for a sequence of 10^3 consecutive updates (insertions/deletions), starting from empty DB. As explained above, we store the first 10 levels of SD_d locally to optimize performance, hence for small database sizes the update time is negligible (less than 0.01ms).

For our tested sizes, MITRA is 1 to 21× faster than SD_d (e.g. for $|DB| = 10^6$ its update time is 1ms while SD_d takes 14ms). We stress that the update time of SD_d is increasing with the number of updates, as more OMAP accesses are necessary. Regarding QOS, we consider only delete operations since they are costlier than insertions. Compared to HORUS our deletion time is, as expected, slightly worse—up to 1.7× slower for the tested sizes (e.g., for $|DB| = 10^8$ QOS takes 281ms and HORUS 233ms). Figure 5.14(c) shows the SD_a update time for 10^3 consecutive updates. For each update, the client has to fetch and merge some of the previously filled indexes, which corresponds to the variable cost shown in the plot. For 10^3 updates, the minimum and maximum observed times are 1μs and 777μs, and the average is 7μs.

Communication size. Figure 5.14(b) shows the update communication size which (not surprisingly) has similar patterns with Figure 5.14(a). For SD_d, due to our optimization (keeping 10 levels locally) the communication size for $|DB| \leq 10^3$ is zero. For $|DB| > 10^3$, the cost of SD_d is larger than MITRA (e.g., $508KB$ vs. $12KB$ for $|DB| = 10^8$). Regarding QOS versus HORUS, the first requires 0.9 to 3.1× more communication than the latter (e.g. for $|DB| = 10^8$ QOS sends 1.6MB, whereas

Horus sends 536KB).

As it is expected, Mitra with local storage has significant more efficient update costs compared to our schemes, since it takes advantage of storing locally the word counters (assuming up to $O(N)$ local storage)—for each update Mitra with local storage requires to compute two PRF evaluations and to store the new value on the server.

5.4.3 Client storage

For all schemes, we store the OMAP stashes and all the keys in K locally at the client. We are interested in measuring the permanent local client storage, in order to ensure it remains reasonably small. Throughout our experiments, the permanent local storage for QOS , Horus , and Mitra was never above 2.5KB, even for $|DB| = 10^6$. With our optimization of storing the 10 smallest levels locally at the client, SD_a and SD_d needed at most $33KB$ and 150KB local client storage, respectively (without this optimization, the corresponding sizes were 400B and 18KB respectively). Recall that we can further reduce the local storage to few bytes ($O(1)$) by storing the stashes on the server and generating the keys from a PRF. However, we consider these sizes *essentially negligible* for modern devices, even for tablets and mobile phones.

5.4.4 Quasi-optimal search performance for variable deletion percentages

Our main motivation for studying DSE with quasi-optimal search time is to avoid paying the cost of past deletions during searches. In all the above experiments, we assume a 10% deletion ratio, rendering the effect of deletions for search negligible. Now, we focus on our new quasi-optimal scheme QOS and we provide experiments for variable deletion ratios.

As is evident from the experiments so far, QOS vastly outperforms the previous state-of-the-art quasi-optimal DSE Horus for searches. In this set of experiments, we compare QOS with SD_d and Mitra (the latter two schemes had better performance for 10% deletions). In this setting, we first insert a fixed number of entries i_w for keyword w and then report the search time after deleting a percentage of i_w between 0-90%. We focus on two cases $i_w = 100$ (*small results*) and $i_w = 20$K (*large results*). Since both SD_d and Mitra have search $\Omega(a_w)$ their performance should worsen as the deletion rate increases. On the other hand, the search time of QOS should be better. Hence, we want to find at which deletion ratio QOS will outperform the others.

Figure 5.15 shows the results for small (a) and large (b) i_w respectively. First, for $i_w = 100$ QOS and Mitra have similar search times since the main bottleneck for both is the OMAP accesses. However, QOS becomes slightly faster and Mitra slightly slower as the deletion ratio increases; the first outperforms the second after roughly 60%. On the other hand, SD_d remains much faster than both of them

throughout the experiment since it does not need to perform OMAP accesses. The results are different for $i_w = 20\text{K}$ in Figure 5.15(b). QOS starts off much slower (as was the case in the experiments above), however, it becomes faster very quickly as deletions increase. It outperforms MITRA at 65% and even SD_d at 80%! The reason is that for large i_w the OMAP cost becomes a small percentage of the search process.

We believe that these results serve as a good indication for the practical potential of schemes with (quasi-)optimal search time while there is still room for improvement. To further support this point, we consider another scenario in which permanent client storage is not a bottleneck and we implement both QOS and MITRA to store the word counter maps locally (avoiding the OMAP overhead) The results are shown in Figures 5.15(c) and (d) for $i_w = 100$ and 20K, respectively. They follow the trends of the corresponding Figures 5.15(a),(b), but the crossover points are moved to the left. QOS becomes better than MITRA for $\sim$40% deletions for $i_w = 100$, and for $\sim$60% for $i_w = 20\text{K}$. Compared to SD_d, it becomes better in both cases at $\sim$80% (in the previous scenario, SD_d was strictly better for $i_w = 100$). In Figure 5.16, we repeated the experiments for the search costs with variable deletion percentage for the crime dataset and we observe similar conclusions.

Chapter 6: SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage

We have discussed in the previous sections that SE-based encrypted databases are quite practical at the expense of well-defined *leakage*. This leakage information includes the *search pattern* (whether a query q has been made in the past or not) and the *access pattern* that consists of the *volume pattern* (number of database tuples contained in the query result) and the *overlapping pattern* (which database tuples, if any, in the result for query q appeared in the result of a previous query).

The aforementioned leakages exposed by SE can be harmful, enabling the recovery of the encrypted database or/and the posed queries. In particular, the works of Islam et al. [76] and Cash et al. [77] were the first to exploit access pattern leakage and prior knowledge about the dataset to recover the queried keywords. Zhang et al. [64] propose file injection attacks for encrypted email applications to improve the recovery rate of queried keywords. Blackstone et al. [78] revisit various assumptions of existing leakage-abuse attacks. For private range search, effective access pattern and volumetric attacks through which the attacker learns the plaintext order and value of encrypted records, without any prior knowledge, have been proposed [79, 80, 81, 82, 83, 84, 85, 86, 87]. This growing body of leakage-abuse at-

tacks has already alerted the community about using SE for implementing encrypted databases [88].

To provably defend against leakage-abuse attacks on SE-based systems one has to (i) use expensive cryptographic tools to eliminate the search/overlapping patterns, i.e., Oblivious RAM (ORAM) [5] (introducing a polylogarithmic search overhead) and (ii) perform worst-case padding (resulting in worst-case linear search time [89] or quadratic index size) for eliminating the volume pattern. Both approaches above incur large overheads leading to quite impractical protocols. We present other, more practical, but less effective defenses in Section 6.1.

In light of the above, we ask in this Chapter whether practical SE primitives can still somehow be used to implement secure encrypted databases. Towards this goal, we propose SEAL[1], a family of new SE schemes with *adjustable leakage* which allow the client to define a trade-off between efficiency and leaked information. We show that hiding *only a few bits* of the search/overlapping/volume pattern significantly reduces the success of existing as well new, even more aggressive, leakage-abuse attacks. At the same time SEAL's practical performance is close to traditional SE. In particular our contributions are as follows:

To better motivate SEAL, we first present new attacks on existing SE-based encrypted databases. In particular, we show that the same inference attacks on DET systems [15] can be used by a persistent adversary to recover the database values in SE-based systems, such as those implementing point queries (e.g., [2, 32]), and group-by and join queries (e.g., [68]). The high-level reason is that after the

[1] **SEAL** stands for **S**earchable **E**ncryption with **A**djustable **L**eakage.

adversary observes a certain number of SE queries in these constructions, tuples with the same values are revealed and therefore frequency information is readily available to the adversary. Even for more robust SE-based range query schemes [2, 34], we present new attacks that can work under certain assumptions about the dataset (see Section 6.2).

We present SEAL(α, x), a family of SE schemes with adjustable leakage. SEAL is based on two other "adjustable" primitives, an adjustable ORAM, parameterized by a value α and an adjustable padding algorithm, parameterized by a value x. The adjustable ORAM, ADJ-ORAM-α, hides only α bits of the access pattern by partitioning the accessed N-sized array into $N/2^\alpha$ regions of 2^α size each and by applying an individual standard ORAM per region. The adjustable padding algorithm, ADJ-PADDING-x, reduces the volume pattern leakage by padding every list to the the closest power of x, leading to a dataset with at most $\log_x N$ distinct sizes. Clearly, larger values for α and x yield slower but more secure SEAL (see Section 6.3).

We use SEAL to build encrypted databases with adjustable leakage. We first present three new construction POINT-ADJ-SE-(α, x) (for point and group-by queries), JOIN-ADJ-SE-(α, x) (for join queries) and RANGE-ADJ-SE-(α, x) (for range queries) that use SEAL(α, x) as black box, instead of plain SE. Finally, we present a more efficient adjustable construction for ranges, RANGE-SRC-SE, that reduces access pattern leakage and volume pattern leakage *implicitly* by modifying an existing constructions [2] and not by using our (more expensive) SEAL(α, x). (see Sections 6.3.4 and 6.3.5).

We evaluate the robustness of our SEAL-based encrypted databases for various values α and x against particularly powerful adversaries that observe the leaked search/overlapping and volume patterns and *have plaintext access to the entire input dataset*. Such strong threat model offers additional credibility to our proposed mitigation techniques. We consider two new attacks. The first is a *query recovery attack* that aims at decrypting the encrypted queries posed by the client. The second is a *database recovery attack* that aims at mapping plaintext values (for the queried attribute) to the tuples of the encrypted database. Note that since SEAL hides some bits of access pattern via ADJ-ORAM, database recovery can be quite challenging (see Section 6.4).

We observe that for all above attacks we can find certain values for α and x that reduce the attacker's success rate significantly while maintaining good performance. For instance we show that if we use SEAL to hide three bits of access pattern while at the same time pad the keyword lists to powers of 4 (thus hiding a few bits of volume pattern as well), we can defend against our powerful attackers only at the expense of an acceptable slowdown from plain SE—around $32\times$.[2]

6.1 Other Relevant Works

Wagh et al. [90] introduces an ORAM with a tunable trade-off between the search/storage efficiency and security. This trade-off is controlled by an (ϵ, δ)-

[2]In Section 6.4, we report for certain parameters of α and x the performance of SEAL compared with the most secure solution (sequential scan) and the one that leaks access and search patterns (SE scheme). We highlight that both sequential scan and SE are **not** competitors of SEAL since they provide different security, but we used those two schemes only as reference points.

differential privacy modification of PathORAM [5]. Their construction could potentially be used as a drop-in replacement in our proposed encrypted database algorithms (instead of our adjustable ORAM). It would be interesting to explore how different choices of ϵ and δ affect the performance of existing leakage-abuse attacks—we leave this as future work.

The works of Cash et al. [77], and Bost and Fouque [91] propose padding techniques for keyword search that can hide a portion of the volume pattern. Unlike our proposed padding in Section 6.3.2, their padding depends on the distribution of the input dataset, which results in leakage even prior to query execution. Similar padding approaches have been also proposed in other areas, e.g., [92] proposes padding approaches for preventing snapshot attacks on deterministically encrypted data and [93] proposes padding for traffic analysis attacks. Bost and Fouque [91] also propose new security definitions for SE aiming at capturing existing leakage abuse attacks. These theoretical definitions could potentially provide some intuition on how we can modify existing schemes in order to make them robust against such attacks.

Recently, Kamara et al. [94] showed how to suppress the search pattern leakage without using ORAM. However suppressing only the search pattern leakage is not enough for mitigating leakage-abuse attacks. Kamara and Moataz [89] showed theoretically how to perform worst-case padding without requiring quadratic index size, while sometimes assuming certain properties for the input dataset, such as a Zipf distribution or highly-concentrated multimaps.

6.2 Encrypted Databases from Searchable Encryption & Attacks

In this section we first show how SE can be used to support various queries on encrypted databases, such as point/group-by/join/range queries and then show various attacks (some existing and some new) on these constructions. Our findings systematically re-establish that using SE to implement encrypted databases [2, 34, 68] is particularly risky when the adversary is persistent and also has access to prior information about the underlying encrypted database (e.g., distribution of first names/gender). For snapshot adversaries that have no prior information about the encrypted database, there could be value in SE-based systems, however these are assumptions that are unlikely to hold in the real world [15, 95].

6.2.1 SE-based Point Queries

The most basic database query is the *point* query for a value v. A point query retrieves all the tuples from table $\mathcal{T}$ that contain value v in attribute x, i.e.,

$$\texttt{SELECT} * \texttt{FROM } \mathcal{T} \texttt{ WHERE } \mathcal{T}.x = v;$$

We can use an SE scheme to implement private point queries (e.g., see Demertzis et al. [2], and Kamara and Moataz [68]) by viewing attribute values as keywords, and database tuples as document identifiers. In this case an SE-based point query will return the encrypted tuples that match this value. We call this scheme POINT-SE. Note that POINT-SE can also be used to implement *group-by* queries (e.g., see Kamara and Moataz [68]), where a client can compute the group-by query through

point queries for all distinct values of attribute x.

Attacks on POINT-SE. When using POINT-SE, the attacker can identify which encrypted tuples have the same value v, after he observes the execution of a query. Finally, after he observes the execution of all queries, the attacker can group the encrypted database tuples by value, and can therefore compute the size of each group. By running a frequency analysis attack or an $\ell_{\mathbf{p}}$-optimization attack (described in Section 2), it is easy to map plaintext values to encrypted tuples. Note that the above attack requires the attacker to see all queries. However, in the case of group-by queries, the very nature of the query reveals all possible point queries, resulting in total leakage exposure with just a single query.

To conclude, observing all possible results from point queries (either one by one or via a group-by query) turns an SE-implemented database into a deterministically-encrypted database, making it vulnerable to simple attacks.

6.2.2 SE-based Join Queries

A fundamental query type for relational databases is the *join* query. A simple join of two tables $\mathcal{T}$ and $\mathcal{R}$ on attribute x returns all pairs of tuples from $\mathcal{T}$ and $\mathcal{R}$ that agree on x, i.e.,

$$\texttt{SELECT} * \texttt{FROM}\ \mathcal{T}, \mathcal{R}\ \texttt{WHERE}\ \mathcal{T}.x = \mathcal{R}.x;$$

A simple approach that allows us to support private join queries using SE is the following: We encrypt $\mathcal{T}$ with a semantically-secure encryption scheme and $\mathcal{R}$ with POINT-SE for private point queries on attribute x. Then we stream all the

tuples of $\mathcal{T}$ to the client. Then the client decrypts each tuple $\mathtt{t}$ in $\mathcal{T}$ and queries the SE index for $\mathcal{R}$ (on attribute x) to retrieve the matching tuples of $\mathcal{R}$. Clearly this approach has high bandwidth since it requires streaming a large number of tuples to the client. We call this scheme JOIN-SE. To address the above bandwidth issue, Kamara and Moataz [68] propose a construction that, in the case of two tables $\mathcal{T}$ and $\mathcal{R}$, precomputes the answers to join queries on each possible attribute x. Then they store with SE a mapping from "keyword" x to the precomputed answer (i.e., pairs of pointers to tuples from $\mathcal{T}$ and $\mathcal{R}$ that have the same value on attribute x). This approach requires both significant amount of storage and setup time. We call this scheme JOIN-SE-PRECOMPUTE.

Attacks on JOIN-SE, JOIN-SE-PRECOMPUTE. It is easy to see that JOIN-SE and JOIN-SE-PRECOMPUTE leak the *encrypted join graph*. That is, for each encrypted tuple $\mathtt{t}$ of $\mathcal{T}$, the respective encrypted tuples $\mathtt{t}'$ of $\mathcal{R}$ that have the same value on x with $\mathtt{t}$ are leaked (if such tuples exist).

We propose a simple attack that recovers the values of the encrypted tuples: Assuming we have access to (part of) the plaintext dataset, we can compute the *plaintext join graph* by connecting with an edge tuples from $\mathcal{T}$ and tuples from $\mathcal{R}$ that have the same plaintext value on attribute x. If all tuples in $\mathcal{T}$ and $\mathcal{R}$ have at least one incident edge the attacker can perform the frequency analysis attack on both $\mathcal{T}$ and $\mathcal{R}$ and recover the plaintext values for the encrypted values of attribute x. In this case JOIN-SE and JOIN-SE-PRECOMPUTE provide exactly the same security properties for joins as more efficient encrypted systems based on de-

terministic encryption (e.g., CryptDB [11]). Otherwise the attack can be performed only on the leaked frequencies and JOIN-SE and JOIN-SE-PRECOMPUTE have potentially less leakage than systems based on deterministic encryption.

6.2.3 SE-based Range Queries

In the case of range queries, we want to retrieve all tuples from table $\mathcal{T}$ that contain value $v \in [l, u]$ in attribute x, i.e.,

$$\texttt{SELECT} * \texttt{FROM } \mathcal{T} \texttt{ WHERE } \mathcal{T}.x \geq l \text{ and } \mathcal{T}.x \leq u;$$

One way to support private range queries is to treat each numeric value of attribute x as a keyword and use SE. Then, private range queries can be supported by transforming the range $[l, u]$ to series of private point queries, i.e., searching for the individual values $l, l+1, \ldots, u-1, u$. We call this scheme RANGE-SE. Many attacks that exploit the overlapping and volume patterns exist against RANGE-SE—see [79, 80, 81, 82, 83, 84]. In general, these attacks first compute an ordering of the encrypted tuples and then retrieve the actual values after observing a certain number of queries.

To address this leakage, Faber et al. [34] and Demertzis et al. [2, 3] have proposed new private range constructions that use SE and are *response-hiding*, in that they do not leak overlaps between different range queries. Their main idea, called LOGARITHMIC-SRC in [2], builds a binary-tree data structure with some extra "internal" nodes (see Figure 6.1) on top of the database. Each leaf corresponds to a value $k \in \{0, 1, \ldots, M-1\}$ (where M is the size of the domain of attribute x)

and stores *all* tuples that have value k at attribute x (i.e., a leaf can store more than one tuples). Data stored in a leaf is also copied to its parents. To answer a range search query, we select the root of the smallest subtree fully covering the query. The above data structure defines a natural key-value relationship, where each tree node is a key with the value being its respective database tuples. This allows us to query the data structure privately using SE.

LOGARITHMIC-SRC yields up to $O(N)$ *false positives* where N is the size of the database table. For example, if the range $[3, 5]$ is being queried in Figure 6.1 and there is a single tuple in the range but the rest of the dataset has value 2, node $N_{2,5}$ will be returned and therefore the response will be the entire dataset. LOGARITHMIC-SRC-I, proposed for this problem [2], maintains two LOGARITHMIC-SRC-type binary trees, one on the domain $\{0, \ldots, M-1\}$ that stores constant-size metadata in the leaves (let us call this tree T_1) and one on the domain $\{0, \ldots, N-1\}$ that stores the actual database tuples in the leaves (one per leaf) sorted by the search attribute (let us call this tree T_2). In particular, for every value of the domain $i \in \{0, \ldots, M-1\}$, T_1 stores the subrange of $\{0, \ldots, N-1\}$ that corresponds to database tuples with value i in T_2. Therefore, a range query $[a, b]$ is transformed into two queries: One range query $[a, b]$ in T_1 that returns information that allows one to reconstruct the range $[a', b']$ of T_2 that contains the desired tuples, and finally one range query $[a', b']$ in T_2 that returns those tuples. This approach brings down the worst-case query cost from $O(N)$ to $O(R + r)$, where R is the size of the queried range (and is due to querying T_1) and r is the size of the result (and is due to querying T_2).

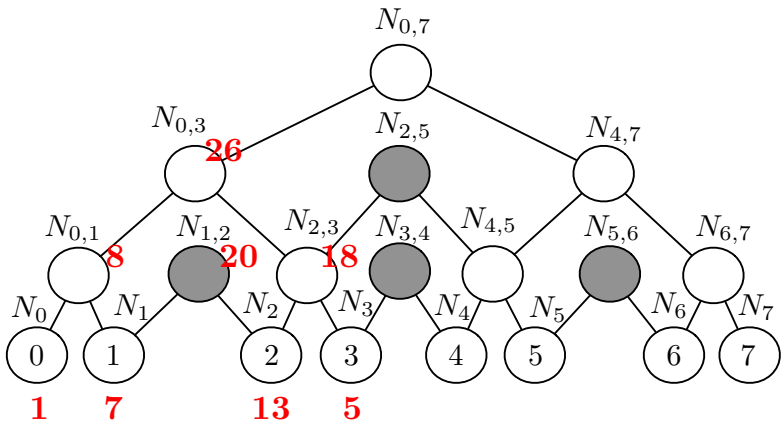


Figure 6.1: LOGARITHMIC-SRC [2, 3] consists of a full binary tree over the domain with an extra internal node between every two cousins. Red values denote the number of tuples each node contains (used for the proposed attack).

Do existing attacks apply? It seems that existing (volumetric) attacks on RANGE-SE [79, 80, 81, 82, 83, 84, 85, 86, 87] do not apply to the above, response-hiding, schemes. However we must note that LOGARITHMIC-SRC and LOGARITHMIC-SRC-I leak the volume pattern of a restricted set of queries and may be vulnerable to new volumetric attacks. In particular, the very recent and concurrent work of Gui et al. [86] proposed new volumetric attacks that can handle cases of missing/spurious queries, and cases that return noisy results. These attacks for missing and noisy queries could potentially be used against LOGARITHMIC-SRC by setting a small window size and treating all volumes from large windows as noise. However, it is not clear how this noise would affect the attack output since the missing queries are not chosen at random as is assumed in [86]. Below, we describe our new attacks tailored to LOGARITHMIC-SRC that could be extended also for LOGARITHMIC-SRC-I.

New attacks on LOGARITHMIC-SRC. The main idea is that if the attacker observes the volumes of all queries, then she could potentially reconstruct the tree

and map encrypted database tuples to plaintext values. For simplicity, let us focus on a LOGARITHMIC-SRC tree with $\mathsf{Dom} = \{0, 1, 2, 3\}$ (and therefore 8 nodes, including the one "extra" internal node—see Figure 6.1). Assume the adversary observes the following sizes of results (he actually sees the respective encrypted tuples as well): 20, 1, 26, 18, 8, 5, 7 and 13. His goal is to map these sizes (and the respective encrypted tuples) to the nodes N_0, N_1, N_2, N_3, N_{01}, N_{12}, N_{23} and N_{03} of the tree. The tuples that map to leaf i will therefore have value i!

To do the mapping the adversary exploits the fact that the size of a parent is equal to the sum of the sizes of its children and therefore sets up 4 linear equations with 8 unknowns $|N_0|$, $|N_1|$, $|N_2|$, $|N_3|$, $|N_{01}|$, $|N_{12}|$, $|N_{23}|$ and $|N_{03}|$. Of course these equations have an infinite number of solutions but the one we are interested in is a permutation of the observed sizes 20, 1, 26, 18, 8, 5, 7 and 13. In our example, due the fact that all pairwise sums are different, there is a unique assignment (up to a mirror arrangement), in particular the assignment $|N_0| = 1$, $|N_1| = 7$, $|N_2| = 13$, $|N_3| = 5$, $|N_{01}| = 8$, $N_{12} = 20$, $N_{23} = 18$ and $N_{03} = 26$. We note here that the described attack would not work in the case where pairwise-sums are not unique (e.g., when all leaves have size 1) but other information could be potentially used in that case. To conclude, this simple attack shows that concealing the overlapping pattern (as LOGARITHMIC-SRC is doing) is not enough for fully defending against range attacks.

Generalization of attack to LOGARITHMIC-SRC-i. Recall that in LOGARITHMIC-SRC-I we maintain two LOGARITHMIC-SRC-type trees:

one for the metadata (T_1) and one for the actual data (T_2). Every leaf in T_1 has size *at most one* since a specific domain value may not be present at all in the database. Thus the above attack that exploits distinct sizes of leaves might not work very well.

However there are still ways to launch an attack. Coming back to Figure 6.1, consider the tree T_1 on the domain $\{0, 1, 2, 3\}$, with the difference that all leaf nodes have size either zero or one. Suppose after all queries have been issued on T_1 the adversary observes only three nodes of size one (and all other nodes have size zero). Looking into this information carefully, one can tell that these nodes have to be either N_0, $N_{0,1}$ and $N_{0,3}$ or N_3, $N_{2,3}$ and $N_{0,3}$ which implies that all database tuples have the same value *and* this value is either 0 or 3. Note that at that point, it will be easy to recover the topology of T_2 since for each range query one node of T_1 and one for T_2 will be accessed together.

The above attacks are not analyzed in full detail since we want to use them mainly as a way to manifest the weaknesses of the Logarithmic-SRC and Logarithmic-SRC-i schemes [2]. We also use them as a motivation to introduce our new RANGE-SRC-SE-(a, x) scheme (see Section 6.3.5). Exploring these attacks against Logarithmic-SRC and Logarithmic-SRC-i in more detail is left as future work.

6.3 SEAL: Adjustable SE & Derived Constructions

Most of the attacks on SE-based encrypted databases that were presented in section 6.2 exploit the leakage of SE such as the *search*, *overlapping* and *volume pattern*. In this section we propose SEAL, a family of new SE schemes with ad-

justable leakage with the hope that these can be used to implement more secure (yet efficient) encrypted databases that withstand leakage-abuse attacks. Our main building blocks are an *adjustable ORAM*, an ORAM that allows one to define the bits of leakage of the index being accessed in a tunable manner, as well a *an adjustable padding* algorithm that adds noise to the actual size of the list being accessed.

6.3.1 Adjustable Oblivious RAM

An adjustable ORAM (ADJ-ORAM-α) is parameterized by a parameter α that defines the number of leaked bits of the accessed memory location ($\alpha = 0$ for a traditional ORAM). We define the ADJ-OramInitialize and ADJ-OramAccess protocols of our ADJ-ORAM-α scheme:

- $(\sigma, \mathsf{EM}) \leftrightarrow$ ADJ-OramInitialize$((1^\lambda, \mathsf{M}, \alpha), \perp)$, takes as input a security parameter λ, a memory array M of n values (without loss of generality lets assume n is a power of 2) $(1, v_1), \ldots, (n, v_n)$, a parameter $\alpha \in \{0, 1, \ldots, \log n\}$ and outputs secret state σ (for client), and encrypted memory EM (for server).

- $((v_i, \sigma), \mathsf{EM}) \leftrightarrow$ ADJ-OramAccess$((\mathtt{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$ is a protocol between the client and the server, where the client's input is the type of operation $\mathtt{op}$ (read/write), an index i and the value v_i—for $\mathtt{op} = \mathtt{read}$ client sets $v_i = \perp$. Server's input is the encrypted memory EM. Client's output consists of the updated secret state σ and the value v_i assigned to the i-th value of M if $\mathtt{op} = \mathtt{read}$ (for $\mathtt{op} = \mathtt{write}$ the returned value is $\perp$). Server's output is the updated encrypted memory EM.

$bit \leftarrow \mathbf{Real}^{\text{ADJ-ORAM-}\alpha}(\lambda)$:

1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $(\sigma_0, \mathsf{EM}_0) \leftrightarrow$ ADJ-OramInitialize$((1^\lambda, \mathsf{M}_0, \alpha), \perp)$.
3: **for** $k = 1$ to q **do** ▷ q: polynomial #queries
4: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5: $\quad ((v_{i_k}, \sigma_k), \mathsf{EM}_k) \leftrightarrow$ ADJ-OramAccess$((\mathsf{op}, i_k, v_{i_k}, \sigma_{k-1}), \mathsf{EM}_{k-1})$.
6: **return** $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$.

$bit \leftarrow \mathbf{Ideal}^{\text{ADJ-ORAM-}\alpha}_{\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha}(\lambda)$:

1: $\mathsf{M}_0 \leftarrow \mathsf{Adv}(1^\lambda)$.
2: $(st_{\mathcal{S}}, \mathsf{EM}_0) \leftarrow$ ADJ-SimOramInitialize$(1^\lambda, \mathcal{L}_1^\alpha)$.
3: **for** $k = 1$ to q **do**
4: $\quad i_k \leftarrow \mathsf{Adv}(1^\kappa, \mathsf{EM}_0, m_1, m_2, \ldots, m_{k-1})$.
5: $\quad (st_{\mathcal{S}}, \mathsf{EM}_k) \leftrightarrow$ ADJ-SimOramAccess$(st_{\mathcal{S}}, \mathsf{EM}_{k-1}, \mathcal{L}_2^\alpha(i_k))$.
6: **return** $bit \leftarrow \mathsf{Adv}(1^k, \mathsf{EM}_0, m_1, m_2, \ldots, m_q)$.

Figure 6.2: ADJ-ORAM-α real-ideal security experiments. With $m_0, m_1, \ldots,$ we denote the messages exchanged at Line 5 of both experiments.

Next, we define the security of ADJ-ORAM-α in the real/ideal game of Figure 6.2 parametrized by leakage functions $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$.

Definition 6 *ADJ-ORAM-α is $(\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha)$-secure if for any PPT adversary* Adv*, there exists a PPT simulator containing algorithms (ADJ-*SimOramInitialize*, ADJ-*SimOramAccess*):*

$$|\Pr[\textbf{\textit{Real}}^{\text{ADJ-ORAM-}\alpha}(\lambda) = 1] - \Pr[\textbf{\textit{Ideal}}^{\text{ADJ-ORAM-}\alpha}_{\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha}(\lambda) = 1]|$$

is at most $neg(\lambda)$, where the above experiments are defined in Figure 6.2 and where the randomness is taken over the random bits used by the algorithms of the ADJ-ORAM-α scheme, the algorithms of the simulator and Adv*.*

The leakages $\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha$ are defined in a manner similar to those of SE, i.e., $\mathcal{L}_1^\alpha(\mathsf{M}) = (n, \alpha)$ and $\mathcal{L}_2^\alpha(i) = id^\alpha(i)$, where $id^\alpha(i)$ returns the α most significant

$(\sigma, \mathrm{EM}) \leftrightarrow$ ADJ-OramInitialize$((1^\lambda, \mathsf{M}, \alpha), \perp)$

1: Let M be in the form $(1, v_1), \ldots, (n, v_n)$ and $\mu = 2^\alpha$.
2: Sample a secret key $k \leftarrow^{\$} \{0,1\}^\lambda$.
3: Let π_k be a PRP: $\{0,1\}^\lambda \times \{0,1\}^{\log_2 n} \to \{0,1\}^{\log_2 n}$.
4: Create $S_1, \ldots, S_\mu$ empty arrays of size $\frac{n}{\mu}$.
5: **for** $i = 1, \ldots, n$ **do**
6: Let ℓ be the integer representation of the α most significant bits of $\pi_k[i]$ and ϕ be the integer representation of the remaining bits of $\pi_k[i]$.
7: $S_{\ell+1}[\phi + 1] = v_i$.
8: **for** $i = 1, \ldots, \mu$ **do**
9: $(\sigma_i, \mathrm{EM}_i) \leftrightarrow$ OramInitialize$((1^\lambda, S_i), \perp)$.
10: Let EM to be $\mathrm{EM}_1, \ldots, \mathrm{EM}_\mu$ and σ to $(\sigma_1, \ldots, \sigma_\mu)$.
11: **return** (σ, EM).

$((v_i, \sigma), \mathsf{EM}) \leftrightarrow$ ADJ-OramAccess$((\mathsf{op}, i, v_i, \sigma, \alpha), \mathsf{EM})$

1: Parse σ as $(\sigma_1, \ldots, \sigma_\mu)$ and EM as $(\mathsf{EM}_1, \ldots, \mathsf{EM}_\mu)$ where $\mu = 2^\alpha$.
2: Let ℓ be the integer representation of the α most significant bits of $\pi_k[i]$ and ϕ be the integer representation of the remaining bits of $\pi_k[i]$.
3: $\ell = \ell + 1$ and $\phi = \phi + 1$.
4: $((v_i, \sigma_\ell), \mathrm{EM}_\ell) \leftrightarrow$ OramAccess$((\mathsf{op}, \phi, v_i, \sigma_\ell), \mathsf{EM}_\ell)$.
5: **return** $(v_i, \sigma, \mathrm{EM})$.

Figure 6.3: ADJ-ORAM-α using any ORAM as a black box.

bits of a random $\log n$-bit alias assigned to tuple (i, v_i). Intuitively, if two queries for index i are made on an ADJ-ORAM-α, the adversary should only figure out that the α most significant bits of the queried index are the same—but nothing else.

Construction of ADJ-ORAM-α. The main idea behind our approach is that the memory array will not be stored in one ORAM, but it will be partitioned into multiple disjoint subsets, each of which will then be stored in a separate smaller ORAM. We use as a black box any secure ORAM= (OramInitialize, OramAccess) to store each subset. Our construction works by building 2^α different ORAMs ORAM_1,...,ORAM_{2^α}, each of which will store a part of M of size $n/2^\alpha$.

One possible way to partition M into these ORAMs would be to determin-

istically assign (i, v_i) based on their location in M, i.e., the first 2^α entries will be stored in ORAM_1, the next 2^α entries will be stored in ORAM_2 and so on. However, this might reveal sensitive information for certain application settings, e.g., if the server knows that M stores v_i in a sorted manner, then accessing ORAM_1 reveals that one of the smallest values in M was accessed. Hence, before performing the partitioning, we randomly permute M using a PRP P over $[1, n]$ (implemented with a small-domain PRP [96, 97, 98]), for which the key k is chosen and stored by the client. Let π_k be the corresponding mapping after k has been chosen. Then, the partitioning of M is performed using the integer representation of the α most significant bits of the permuted index and the remaining bits of $\pi_k(i)$ correspond to the index $\pi_k(i)$ of tuple (i, v_i) inside the small ORAM. Our construction is given in Figure 6.3.

Theorem 7 *Assuming* (OramInitialize, OramAccess) *is a secure ORAM and* π_k *is a secure PRP, then* ADJ-ORAM-α *presented above is* $(\mathcal{L}_1^\alpha, \mathcal{L}_2^\alpha)$*-secure, according to Def. 6.*

Proof 8 *The ORAM scheme used is secure and therefore we use its algorithms* SimOramInitialize *and* SimOramAccess. *In particular, the ADJ-*SimOramInitialize *takes as an input* $\mathcal{L}_1^\alpha = (n, \alpha)$ *and the security parameter* λ*, and it creates* $\text{EM}_1, \ldots \text{EM}_\mu$ *and* $\sigma_1, \ldots, \sigma_\mu$ *using* SimOramInitialize$(1^\lambda, \frac{n}{\mu})$ *for* $\mu = \frac{n}{2^\alpha}$. *The ADJ-*SimOramAccess *takes as an input* $id^\alpha(i)$*, from* $\mathcal{L}_2$ *leakage, which determines in which encrypted memory* EM_i *must be accessed, and performs a random access using* SimOramAccess(σ_i, EM_i). *Then, the simulator properly updates* EM_i *and* σ_i.□.

Performance and leakage of ADJ-ORAM-α. The higher the value of α is, the more efficient ADJ-ORAM is (ORAM is applied on a smaller parts of the array) and the larger the leakage becomes (more accesses will be made on the same small parts of the array). Concretely, if we assume that the ORAM used as a building block has $T(n)$ access overhead (e.g., $T(n) = O(\log n)$ for the most efficient ORAM [99]), then ADJ-ORAM-α has an improved $T(n/2^{\alpha})$ overhead. In Section 6.3.3 we discuss how ADJ-ORAM-α can be instantiated using [5] and oblivious data structures [41] and we provide a more concrete performance analysis.

6.3.2 Adjustable Padding

In this section we propose *adjustable padding*, another primitive that will help us build more secure SE schemes. Recall that existing SE schemes leak the query result size, i.e., $|\mathcal{D}(w)|$. In particular, in a dataset with size N a keyword list can have N different sizes. One way to eliminate this leakage is by *padding* all the keyword lists $D(w)$ to the same size N (worst-case padding). However, this would introduce a prohibitive storage/search overhead. To avoid this overhead, one could pad to the closest power of two, forcing the adversary to observe at most $\log N + 1$ sizes—leaking $\log \log N + 1$ bits, at most doubling the search and storage overhead.

Our proposal is a generalization of the above idea. Our padding can be parameterized by a value x that defines the number of different sizes (which are exactly $\lceil \log_x N \rceil + 1$) that the adversary can observe. Our padding algorithm works as follows (see Figure 6.4). Given a keyword list $\mathcal{D}(w)$ of size, we find the integer i

$\mathcal{D} \leftarrow$ ADJ-Padding$(x, \mathcal{D})$

1: $N = |\mathcal{D}|$.
2: **for** each keyword w in $\mathcal{D}$ **do**
3: Find the smallest i: $x^{i-1} < |\mathcal{D}(w)| \leq x^i$.
4: Pad $\mathcal{D}(w)$ with $x^i - |\mathcal{D}(w)|$ dummy values.
5: Pad $\mathcal{D}$ with dummy records so that the total size is $x \cdot N$.
6: **return** the padded dataset.

Figure 6.4: ADJ-Padding-x leading to $\log_x N$ different sizes.

such that $x^{i-1} < |\mathcal{D}(w)| \leq x^i$. Then we pad the list $\mathcal{D}(w)$ with $x^i - |\mathcal{D}(w)|$ dummy entries. Note that this padding strategy can increase the space and search overhead by a factor of x and yields leakage of $\log \log_x N + 1$ bits! In other words the larger x is, the less efficient the scheme becomes and the less leakage the adversary observes. We note here that for simulation purposes, after all lists are padded, our algorithm pads the dataset to a total of $x \cdot N$ entries so that to avoid leaking any information about the dataset.

We note here that padding techniques have been used before for concealing the size of the accessed result (e.g., see Cash et al. [77] and Bost and Fouque [91], as well as Lacharite et al.[92] and Liberatore et al.[93]). However, these approaches depend on the distribution of the input dataset, which leads to more leakage, even prior to query execution. Instead our padding algorithm is *distribution-agnostic* and can thus be simulated only by knowing the size of the dataset N and the padding parameter x.

$\underline{(st_{\mathsf{C}}, \mathcal{I}) \leftarrow \textsc{Setup}(1^\lambda, \mathcal{D})}$

1: Let $\mathcal{D}$ be the input dataset and let $\mathbf{W}$ be the set of keywords in $\mathcal{D}$.
2: $\mathcal{D} \leftarrow \mathsf{ADJ\text{-}PADDING}(x, \mathcal{D})$. ▷ Parameter x is public.
3: Let M be an array of N entries storing (w, id) pairs of $\mathcal{D}$ in lexicographic order and i_w be the index of w's first occurrence in M.
4: $(T, \sigma_{\text{odict}}) \leftarrow \text{ODictSetup}(1^\lambda, N)$.
5: **for** all $w \in \mathbf{W}$ **do**
6: Let $cnt_w = |\mathcal{D}(w)|$.
7: $(\sigma_{\text{odict}}, T) \leftrightarrow \text{ODictInsert}((w, i_w || cnt_w, \sigma_{\text{odict}}), T)$.
8: $(\sigma_{\text{oram}}, \text{EM}) \leftarrow \text{ADJ-OramInitialize}(1^\lambda, \mathsf{M}, \alpha)$. ▷ Parameter α is public.
9: $st_{\mathcal{C}} = (\sigma_{\text{oram}}, \sigma_{\text{odict}})$ and $\mathcal{I} = (\mathsf{EM}, T)$.
10: **return** $(st_{\mathcal{C}}, \mathcal{I})$.

$\underline{((\mathcal{X}, st_{\mathcal{C}}), \mathcal{I}) \leftrightarrow \textsc{Search}((st_{\mathcal{C}}, w), \mathcal{I})}$

1: Parse $\mathcal{I}$ as (EM, T) and $st_{\mathcal{C}}$ as (σ_{odict}, σ_{oram}) and let $\mathcal{X}$ be empty.
2: $((\texttt{value}, \sigma_{\text{odict}}), T) \leftrightarrow \text{ODictSearch}((w, \sigma_{\text{odict}}), T)$.
3: Parse $\texttt{value}$ as $(i_w || cnt_w)$.
4: **for** $i = i_w, \ldots, i_w + cnt_w$ **do**
5: $((v_i, \sigma_{\text{oram}}), \mathsf{EM}) \leftrightarrow \text{ADJ-OramAccess}((read, i, \perp, \sigma_{\text{oram}}, \alpha), \mathsf{EM})$.
6: $\mathcal{X} \leftarrow \mathcal{X} \cup v_i$.
7: **return** $(\mathcal{X}, st_{\mathcal{C}}, \mathcal{I})$.

Figure 6.5: Our SEAL(α, x) scheme using ADJ-ORAM-α, ADJ-PADDING-x, and an oblivious dictionary as black boxes.

6.3.3 SEAL

We now present SEAL(α, x), our adjustable SE construction that uses ADJ-ORAM-α, ADJ-PADDING-x and an oblivious dictionary ODict described in Section 2 as black boxes.

We recall that parameter α is defined in the range $[0, \log N]$ and that for $\alpha = 0$ all the search/overlapping pattern bits are protected, and for $\alpha = \log N$ all bits are leaked. Also for larger x values, less volume pattern bits are leaked—e.g., for value $x = N$ no volume pattern bits are leaked.

Construction of SEAL(α, x). SEAL(α, x) is defined similarly with SE (see Sec-

tion 2) and has algorithms/protocols Setup and Search. Our construction is described in Figure 6.5.

SEAL's setup takes as input dataset $\mathcal{D}$. Parameters α and x are considered public and we do not provide them as input explicitly. First, it uses ADJ-PADDING$(x, \mathcal{D})$ in order to transform $\mathcal{D}$ to a new dataset with at most $\log_x N + 1$ distinct results sizes (see Line 2 of setup). Then, it sorts all the (w, id) pairs in lexicographical order (see Line 3 of setup) and places them sequentially in a memory array M which is then given as input to the ADJ-OramInitialize algorithm (see Line 8 of setup). The sorting guarantees that all (w, id) for the same keyword w will be placed in consecutive memory locations. All entries for w can then be retrieved if one knows the index of the first appearance of w and the size of the padded list $|\mathcal{D}(w)|$. For every keyword w, this information is stored in an oblivious dictionary T (see Line 7 of setup).

SEAL's search takes as input the queried keyword w, client's secret state st_C and the encrypted index $\mathcal{I}$, which contains the small oblivious memories $\mathsf{EM}_1, \ldots$ as well as the oblivious dictionary T. For a given queried keyword w, the client first performs an access to the oblivious dictionary to retrieve the index of the first appearance of w in M and the padded result size (cnt_w) (see Lines 2-3 of search). Then, it performs cnt_w accesses in the ADJ-ORAM-α in order to retrieve the result $\mathcal{X}$ (see Lines 4-7 of search). Note that, due to padding, $\mathcal{X}$ may contain "dummy" records which will be filtered out by the client afterwards.

Leakage definition for SEAL(α, x). SEAL(α, x) is secure according to the stan-

dard SE/OSE definition described in Section 2 with the following leakage functions

$$\mathcal{L}_1^{\alpha,x}(\mathcal{D}) = (N, \alpha, x) \text{ and } \mathcal{L}_2^{\alpha,x}(\mathcal{D}, w) = \mathcal{D}_\alpha^x(w)\,,$$

where $\mathcal{D}_\alpha^x(w)$ contains the α most significant bits of the aliases of the document identifiers in the padded list $\mathcal{D}(w)$ as output by algorithm ADJ-PADDING$(x, \mathcal{D})$. For the rest of the chapter we simply denote these leakages as $\mathcal{L}_1$ and $\mathcal{L}_2$.

Theorem 8 *Assuming that* ODict *is a secure oblivious data structure according to [41] (Def. 1) and* ADJ-ORAM-α *is secure according to Def. 6, then* SEAL(α, x) *is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure according to Def. 1.*

Proof 9 *ADJ-ORAM-α is secure—our proof uses simulator algorithms ADJ-*SimOramInitialize *and ADJ-*SimOramAccess*. The security parameter λ is given. The* SimSetup *takes as an input $\mathcal{L}_1 = (N, \alpha, x)$.* SimSetup *initializes $(T, \sigma_{odict}) \leftarrow$* ODictSetup$(1^\lambda, N)$ *and it inserts N random entries of the form $(w, i_w||cnt_w)$ in the oblivious dictionary T using* ODictInsert*. Then, it computes $N' = x \cdot N$. Finally, it uses ADJ-*SimOramInitialize$(1^\lambda, N', \alpha)$ *to create the encrypted memory* EM *and state σ_{oram}. The* SimSearch *algorithm takes as an input $\mathcal{L}_2$ and performs one random access in the oblivious dictionary T using* ODictSearch*, and calls $|\mathcal{D}_\alpha^x(w)|$ times the ADJ-*SimOramAccess *with input the α-bit identifiers in $\mathcal{D}_\alpha^x(w)$ ($\mathcal{D}_\alpha^x(w)$ has the required leakage for ADJ-*SimOramAccess*). Then, the simulator updates* EM$, T$ *and the states σ_{odict}, and σ_{oram}.* □

Asymptotic performance. Let $(T(n), C(n), S(n))$ be the access complexity, client-space complexity and server-space complexity respectively of the underlying ORAM used and let $(t(n), c(n), s(n))$ be the access complexity, client-space complexity and server-space complexity respectively of the underlying oblivious dictionary used. The server space required is always $S(x \cdot N) + s(N)$. Now, assuming the client keeps, along with the oblivious dictionary state, the ORAM states locally, the search complexity for a keyword w is

$$t(N) + x \cdot |\mathcal{D}(w)| \cdot T\left(\frac{x \cdot N}{2^{\alpha}}\right)$$

and the client space is $2^{\alpha} \cdot C(x \cdot N/2^{\alpha}) + c(N)$. Assuming the client does not keep ORAM states locally and just downloads and re-encrypts to the server, the search complexity for w is

$$t(N) + x \cdot |\mathcal{D}(w)| \cdot \max\left\{T\left(\frac{x \cdot N}{2^{\alpha}}\right), C\left(\frac{x \cdot N}{2^{\alpha}}\right)\right\}$$

and the client space is just $c(N)$. Whether one chooses to store the local states locally or outsource them depends on the parameter α. For small values of α it is better to keep them locally, while for larger values of α it might worth outsourcing.

Implementing ADJ-ORAM-α. We implement each small ORAM in ADJ-ORAM-α with Path-ORAM [5]. Recall that the cost of Path-ORAM for accessing n blocks of size B is $B \log n$ for accessing the path and $O(\log^3 n)$ for recursively updating the position map. In our case we apply Path-ORAM on $N/2^a$ blocks of size around

$2 \log N$ bits ($\log N$ bits for storing keyword w and $\log N$ bits for storing the id) and therefore our total cost is $O(\log N \log(N/2^a) + \log^3(N/2^a))$.

Implementing SEAL(α, x). For SEAL(α, x), apart from ADJ-ORAM-α as described above, we also use an oblivious dictionary ODICT (for storing $i_w||cnt_w$) implemented with an oblivious AVL tree [41] (this requires $b \log^2 N$ additional additive cost where b is the bitsize of $i_w||cnt_w$). In case the number of keywords/attributes $|\mathbf{W}|$ in small, we choose to keep the dictionary locally—this requires around $3|\mathbf{W}| \log N$ bits which in practice is a few megabytes and is a common assumption in Dynamic SE [28, 65, 66, 71]. Our experiments in the next section assume the dictionary is kept locally. Note that even if we do not keep the dictionary locally, we only require one oblivious access to it per query w. This is most of the times subsumed by the required $|\mathcal{D}(w)|$ ADJ-ORAM-α queries, especially when $|\mathcal{D}(w)|$ is large (e.g., $\Omega(\log^2 N)$). In any case we can always reduce the above cost with an adjustable oblivious dictionary at the expense of leaking α bits of the search pattern. Finally, in case the worst-case overhead of SEAL(α, x) becomes higher than sequential scan (which has no leakage), we perform a sequential scan.

6.3.4 New Constructions for Point/Join Queries

In Section 6.2 we presented/reviewed three constructions for point and join queries on encrypted databases that use *SE as a black box*: (i) POINT-SE, a construction for point queries on encrypted data; (ii) JOIN-SE and JOIN-SE-PRECOMPUTE, two constructions for join queries on encrypted data.

Our proposed new constructions reduce the leakage of the above constructions by using SEAL(α, x), instead of simple SE. By doing this replacement we have the following constructions, for various parameters of α and x,

1. POINT-ADJ-SE, and 2) JOIN-ADJ-SE.

Note that JOIN-ADJ-SE can be instantiated either by using JOIN-SE or JOIN-SE-PRECOMPUTE as basis. We denote also that POINT-ADJ-SE can also be used in the case of group-by queries.

6.3.5 New Constructions for Range Queries

The first adjustable construction that we propose for range queries, RANGE-ADJ-SE-(a, x), is based on the "naive" construction RANGE-SE from Section 6.2.3, where instead of simple SE we use SEAL(a, x).

Our second construction, RANGE-SRC-SE-(a, x) comprises two modifications of LOGARITHMIC-SRC-I [2] so that the potential attack presented in Section 6.2.3 is mitigated. Recall the attack works by exploiting volumes exposed by tree T_1 (the tree T_1 stores metadata required to search tree T_2).

Our first modification of LOGARITHMIC-SRC-I is a simple one: Instead of outsourcing tree T_1 using SE, keep tree T_1 locally unencrypted and therefore previously exposed volume information will not be available. The only downside is the $O(|\mathbf{W}|)$ client storage that is required to store T_1, where $\mathbf{W}$ is the set of values of the range attribute. In practice this storage is minimal, e.g., none of the ranges of the attributes shown in Table 6.1 of our evaluation exceed 1MB. (Of course, if

strictly necessary, we can outsource tree T_1 to the server via an oblivious dictionary without any leakage, increasing the search time by a polylog factor.)

RANGE-SRC-SE-(α, x). However, the above modification addresses the leakage only in T_1. But T_2 can also leak information. For example, (a) if the same tree node is accessed twice, there is nonzero probability that the same range is being queried, and (b) the result size (or an upper bound of it) is leaked from accessing T_2. To reduce the effect of leakages (a) and (b), one could reduce the number of sizes observed by the adversary by implementing the encrypted index for T_2 using SEAL(α, x) instead of simple SE.

Our second modification that yields our final scheme RANGE-SRC-SE-(α, x) does almost that, but it does *not* use ADJ-Padding for reducing the volume pattern leakage—this would blow up the space to $O(xN \log(xN))$. Instead RANGE-SRC-SE-(α, x) reduces the number of sizes that are being observed to $\log_x N + 1$ by storing only as many *equally distributed* levels from T_2. E.g., for $x = 2$ all levels are stored, for $x = 4$ half of the levels are stored, while for $x = 16$ one fourth of the levels are stored. Note that by this approach the search complexity is $O(x \cdot r)$ and the space is $O(N \log_x N)$.

6.4 Evaluation Against Attacks

To benchmark the effectiveness of our proposed adjustable constructions POINT-ADJ-SE, JOIN-ADJ-SE and RANGE-SRC-SE, we could use existing state-of-the-art leakage-abuse attacks [76, 77, 79, 80, 81, 82]. However, these attacks are

very sensitive to the *exact* overlapping or volume pattern (e.g., for ordering the records in range queries), which is not available in our adjustable constructions. We introduce instead a new class of attacks where the adversary tries to work with only the available bits of leakage, and at a high level, tries to guess the rest of the bits. Also, our adversary is *quite powerful*, having plaintext access to the input dataset. We stress that this is a "heavy" benchmark that already covers known attacks [76, 77, 79, 80, 81, 82]. This is because if our adjustable constructions reduce the success rate of such a powerful attacker, a more realistic attacker with partial knowledge of the dataset would perform even worse (assuming the same attack strategy is followed). We now describe the attacker model in detail.

6.4.1 Attacker Model

Our model considers a single-client setting (we do not support a multi-client scenario with multiple parties accessing the data). We assume that our adversary: (i) is the system provider that hosts the encrypted database (including the encrypted index) and performs the encrypted query execution; (ii) is honest-but-curious (i.e., tries to infer information during the execution of the protocol, but does not deviate from the protocol, e.g., to give a "tampered" answer); (iii) has full visibility of the server-side execution and memory; (iv) acquires all the possible leaked information from query execution—observing all possible queries at least once; (v) has access to 100% of the plaintext database. Our adversary has two goals:

1. First, to perform a *query recovery attack*, namely decrypting the client en-

crypted queries;

2. Second, to perform a *database recovery attack*, that requires to map plaintext values (for the queried attribute) to the tuples of the encrypted database.

We stress that this a strong attacker model, one that we believe is beyond most real-world adversaries' capabilities. This was a deliberate design decision as our main goal is to evaluate our proposed mitigation techniques against a strong adversary. On the other hand, our analysis does not capture cases where the attacker has information about the query distribution. Note here that a database recovery attack in the case of SE ($\alpha = \log N$) is trivial, since the identifiers of the encrypted records reveal the desired mapping to the plaintext records directly. This task becomes more challenging for smaller values of α where this information is not given in its entirety. In addition, note that the database recovery attack becomes also trivial if SEAL does not re-randomize or assign new tuple ids to encrypted tuples; which is not the case in SEAL (see Line 6 of the used ADJ-ORAM-α). For our experiments, we define the query recovery success rate QR_{SR} as the ratio of the number of correctly decrypted queries over the total number of considered queries. We also define the database recovery success rate DR_{SR} as the ratio of the number of encrypted tuples that have been correctly mapped to the plaintext tuples.

6.4.2 Experimental Setup

Our experiments were conducted on a 64-bit machine with an Intel Xeon E5-2676v3 and 64 GB RAM. We utilized the JavaX.crypto and the bouncy castle

$QR_{SR} \leftarrow \mathsf{QueryRecoveryAttack}(\mathcal{T}, \{t_q, |q|\}_{q \in Q})$

Input: Plaintext tuples $\mathcal{T}$ and tokens t_q along with their volumes $|q|$.
Output: The success rate QR_{SR} of the attack.

1: Set $\mathcal{T} \leftarrow \mathsf{ADJ\text{-}Padding}(x, \mathcal{T})$.
2: Set CORRECT $= 0$.
3: **for** each token t_q **do**
4: Choose q' at random from the set $\{q' : |\mathcal{T}(q')| = |q|\}$.
5: Remove q' from $\mathcal{T}$.
6: **if** q' is the correct decryption for t_q **then**
7: CORRECT++.
8: **return** CORRECT$/|Q|$.

Figure 6.6: Query Recovery Attack for Point Queries.

$DR_{SR} \leftarrow \mathsf{DatabaseRecoveryAttack}(\mathcal{T}, enc(\mathcal{T}), \{t_q, S_q\}_{q \in Q})$

Input: Plaintext tuples $\mathcal{T}$, encrypted tuples $enc(\mathcal{T})$ and tokens t_q along with respective set S_q of encrypted tuples (and their α-bit identifiers).
Output: The success rate DR_{SR} of the attack.

1: Set $\mathcal{T} \leftarrow \mathsf{ADJ\text{-}Padding}(x, \mathcal{T})$.
2: Set CORRECT $= 0$.
3: **for** each pair (t_q, S_q) **do**
4: Choose q' at random from the set $\{q' : |\mathcal{T}(q')| = |S_q|\}$.
5: **for** each encrypted tuple $e \in S_q$ **do**
6: Let id be the α-bit identifier of e.
7: Choose at random a tuple t from $enc(\mathcal{T})$ that has id as the first α bits of its identifier.
8: Remove t from $enc(\mathcal{T})$.
9: **if** encrypted tuple t has value q' at the queried attribute **then**
10: CORRECT++.
11: Remove q' from $\mathcal{T}$.
12: **return** CORRECT$/\sum |S_q|$.

Figure 6.7: Database Recovery Attack for Point Queries.

library [53] for the cryptographic operations. Our java implementation does not use hardware supported cryptographic operations. However, this does not affect our conclusions. The use of hardware supported cryptographic operations can further improve the absolute time for construction and search, but it will not affect the

comparison for different parameters α and x. We consider the following two datasets in our experimental evaluation. For attacking POINT-ADJ-SE-(α, x), we use a real dataset consisting of 6,123,276 tuples with 22 attributes of reported incidents of crime in Chicago [46]. For attacking POINT-ADJ-SE-(α, x), JOIN-ADJ-SE-(α, x), and RANGE-SRC-SE-(α, x), we used the TPC-H benchmark [49] with scaling factor 0.1 which is widely used by the database community[3]. TPC-H consists of eight separate tables (PART, SUPPLIER, PARTSUPP, CUSTOMER, NATION, LINEITEM, REGION, ORDERS). Our attacks take as input the leakage of all possible queries (worst-case leakage). The same attacks can be run with less queries, leading to lower success rate. When evaluating the performance of SEAL(α, x) we store the oblivious dictionary locally. We denote with $x = \perp$ the lack of padding, where the attacker can observe up to N distinct result sizes.

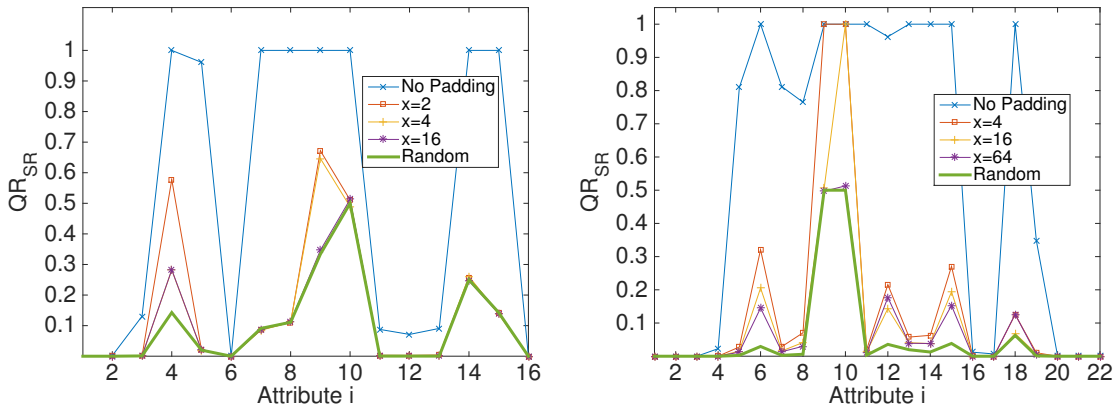
6.4.3 Attacking POINT-ADJ-SE

We evaluate the effectiveness of POINT-ADJ-SE-(α, x) against our new query/database recovery attacks. In both attacks we consider one attribute of one table at a time. Our *query recovery* attack (see Figure 6.6) is very simple and uses only volume pattern leakage. Having access to the plaintext table $\mathcal{T}$, the adversary computes the new padded table for the queried attribute (Line 1 in Figure 6.6) using the padding parameter x. Now, for a given encrypted query q with size $|q|$ the adversary uses $\mathcal{T}$ to find the candidate plaintext values which have size $|q|$, and

[3]We do not provide an evaluation for group-by queries since the results are identical to those for point queries (after observing all the distinct queries).

chooses one of them at random (see Line 4 in Figure 6.6). Note that the higher the value of x is, the larger the set of possible values in Line 4 is therefore reducing the success rate of the attack. The *database recovery* (see Figure 6.7) works as follows. First the adversary decrypts which keyword we are querying, as before—say this keyword is q'. Now, the goal is to map the value q' to the correct encrypted tuples in $enc(\mathcal{T})$, where $enc(\mathcal{T})$ is the encrypted database produced by the SETUP algorithm of SEAL. The adversary knowing from $\mathcal{L}_2$ leakage the α-bits of each returned encrypted tuple, chooses at random for each of them one tuple from $enc(\mathcal{T})$ with same α bits as prefix and maps q' to this tuple. Finally, the adversary removes the chosen tuples t from $enc(\mathcal{T})$. The adversary is successful if after this process the encrypted tuple t has value q' at the queried attribute. Clearly, the smaller α is, the more bits the adversary will have to guess (the larger the set of tuples with same α bits as prefix is) and therefore the less successful the attack is going to be.

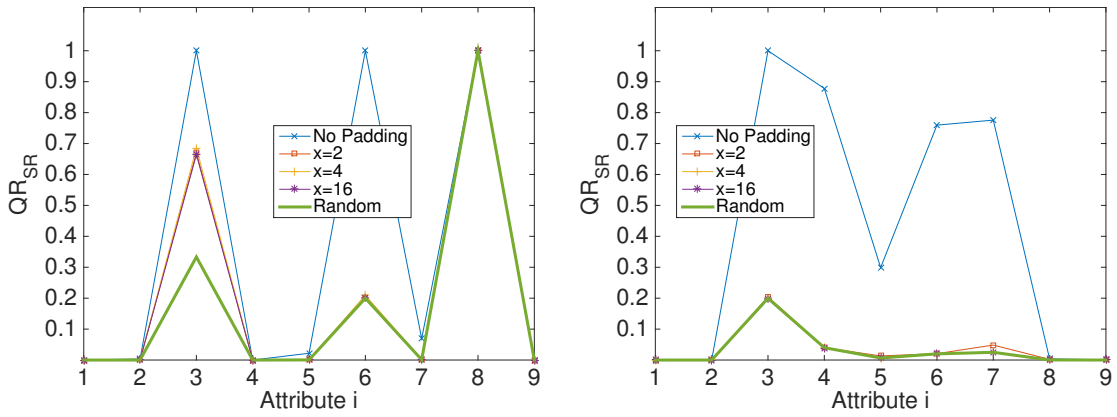
Query recovery attack evaluation. Figures 6.8(a), and 6.8(b) show the evaluation of POINT-ADJ-SE-(α, x) against the query recovery attack. We only vary x since α does not affect the effectiveness of the attack. Figure 6.8(a) demonstrates the evaluation for the LINEITEM table (TPC-H), while Figure 6.8(b) presents the results for the Crime dataset. In all figures, we report the attacker's query recovery success rate if she just maps encrypted queries to plaintext values at random, i.e., $1/|\mathbf{W}|$—ideally, the success rate of our attack should be as close as possible to this "Random" approach. In Figure 6.8(a), for $x = 2$ (only a $2\times$ overhead in search time and storage), we see that our scheme forces the attacker to perform very close to "Random" for 14 out of 16 attributes. We observe that QR_{SR} for attribute 8 is close



(a) LINEITEM (TPC-H) (b) Crime Dataset

Figure 6.8: Query Recovery Attack against POINT-ADJ-SE for various x.

to Random for $x = 16$, while for attribute 4 greater values of x are needed. Let us look why this is the case for, say, attribute 8: There are only three values that can be queried with highly-skewed result sizes $|q_1|$ =1, $|q_2|$ =1,000 and $|q_3|$ =100,000. Therefore the larger the number of padded sizes is, the more likely it is that each q_i will be mapped to a distinct padded size, allowing the attacker to still distinguish them. We observe similar patterns for the tables of TPC-H and we report the results for tables ORDERS and PART in Figure 6.9.



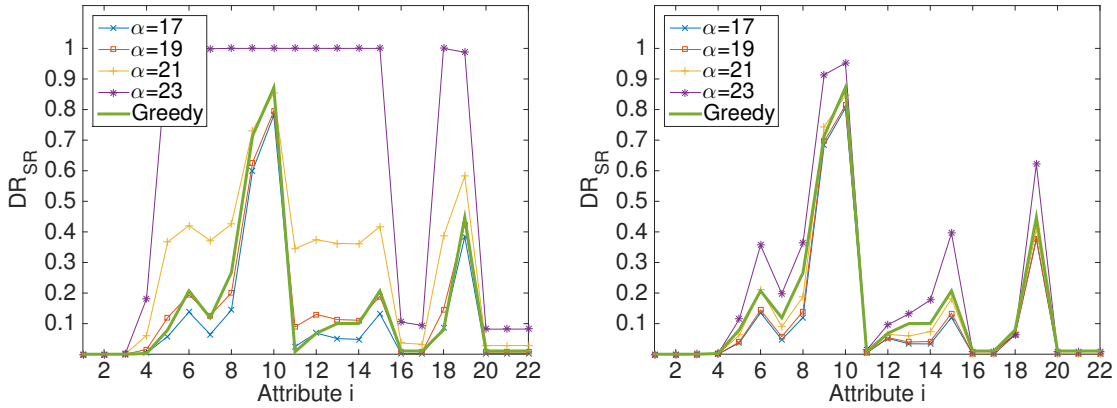
(a) ORDERS (TPC-H) (b) PART (TPC-H)

]

Figure 6.9: Query Recovery Attack against POINT-ADJ-SE for various x.

In Figure 6.8(b) we repeat the same experiment for the 22 attributes of the crime dataset, and we observe that in 17 out of 22 attributes for $x = 4$ (up to 4× performance degradation) the attacker's QR_{SR} significantly drops and is close to the Random approach. For attributes $6, 8, 10, 12, 15$ greater values of x are needed again due to the small number of values that these attributes have. Finally, we observe that in attributes 15 and 18, QR_{SR} is higher for $x = 64$ than for $x = 4$, which is counterintuitive. This is because the query sizes of the values in these attributes are distributed in a way that for $x = 4$ there are less distinct sizes than for $x = 64$.

Database recovery attack evaluation. The database recovery attack is based on the query recovery one. Thus, due to lack of space we focus on the 22 attributes of the crime dataset in which QR_{SR} is higher than the one in the TPC-H dataset. Figure 6.10 shows the attacker's success rate for the database recovery attack (DR_{SR}) for $\alpha = (17, 19, 21, 23)$ ($\alpha = 23$ corresponds to SEAL$(\log N, x)$) and for $x = \perp$ and $x = 2$. Recall that in our threat model the attacker has plaintext access to the input dataset, so for the database recovery attacks we report as a reference point a greedy strategy that the adversary may follow, in which she maps all encrypted tuples to the most frequent plaintext value (guessing heuristically). E.g., for a binary attribute if the most frequent value appears in the 70% of the tuples/tuple-ids then the adversary achieves $DR_{SR} = 70\%$ by following the greedy strategy. Ideally, the goal is to find α as close as possible to $\log N$ and the smallest possible value of x, while DR_{SR} is below the greedy strategy. As is shown in Figure 6.10 for $\alpha = \log N - 2 = 21$ and $x = 2$ the attacker's success rate is always below the success rate of the greedy strategy. In Figure 6.11, we provide a more detailed evaluation



(a) $x = \perp$ (b) $x = 2$

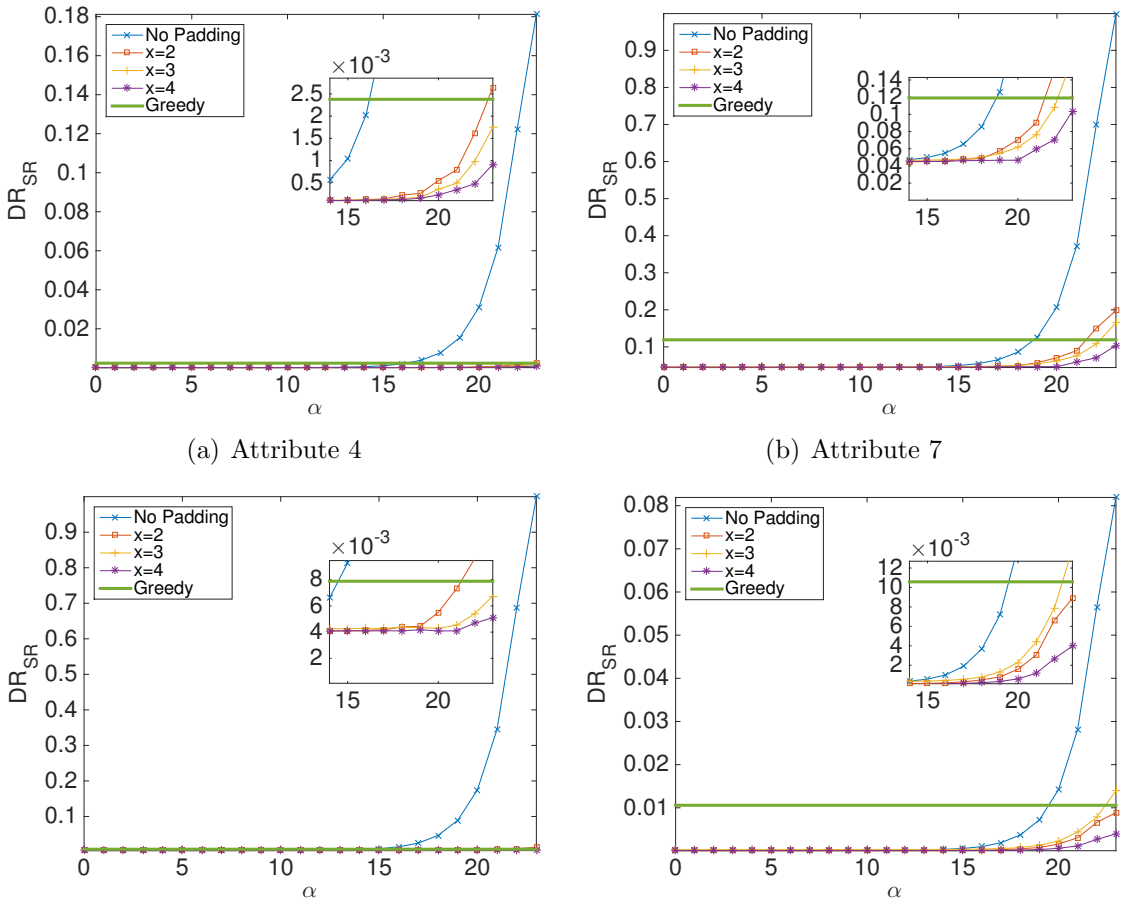
Figure 6.10: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. We show all attributes.

for 4 specific attributes of the crime dataset for $\alpha \in [0, \log N]$ and $x = \perp, 2, 3, 4$.

6.4.4 Attacking JOIN-ADJ-SE

We evaluate the effectiveness of JOIN-ADJ-SE-(α, x) using the database recovery attack proposed for point queries (see Figures 6.7). Since the database schema and the size of each table are usually not considered private information, we do not consider join query recovery attacks.

Attack evaluation. Figure 6.12 demonstrates the database recovery attack for foreign-key join queries. We consider foreign-key joins between tables (i) SUPPLIER and NATION—Figure 6.12(a), and (ii) CUSTOMER and NATION; the TPC-H benchmark contains only foreign-key joins. We observe in Figure 6.12(b) the DR_{SR} for $\alpha = [0, \log N]$, and $x = \perp, 2, 3, 4$. For $\alpha = 0$ and $x = \perp$, DR_{SR} is 65% in Figure 6.12(a) and 97% in Figure6.12(b), but for $\alpha = \log N - 1$ and $x = 2$, DR_{SR} drops below 6%. We conducted all the possible foreign-key joins and we observe the same pattern.



(a) Attribute 4

(b) Attribute 7

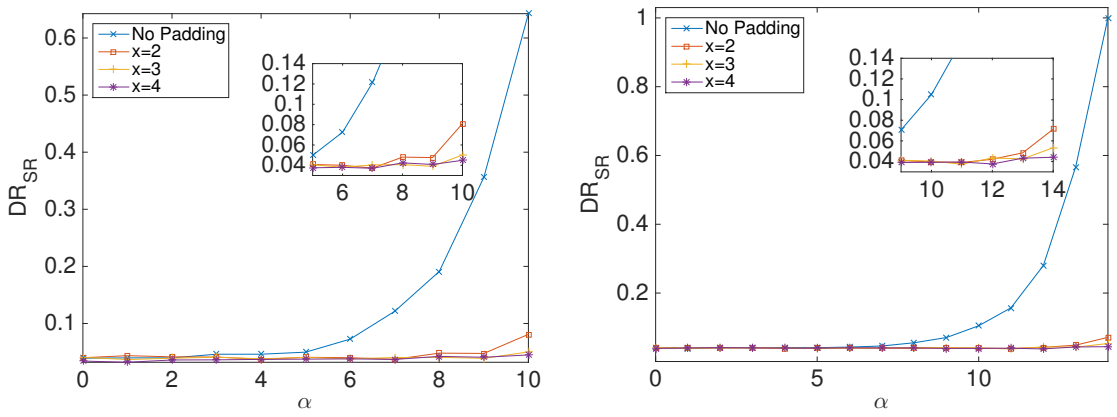
(c) Attribute 11

(d) Attribute 20

Figure 6.11: Database Recovery Attack against POINT-ADJ-SE for the Crime Dataset. Attributes 4,7,11,20.

6.4.5 Attacking RANGE-SRC-SE

We evaluate the effectiveness of RANGE-SRC-SE-(α, x) scheme for various x against slightly modified versions of the attacks for point queries (Figures 6.6 and 6.7). In particular in Line 2 of both Figure 6.6 and 6.7, we do not perform padding but we recreate T_2 in plaintext with only $\log_x N + 1$ evenly distributes levels. We report as a baseline a scheme that does not perform padding but hides the entire overlapping pattern leakage. For the case of query recovery attack we set $\alpha = \log N$



(a) SUPPLIER ⋈NATION (b) CUSTOMER ⋈NATION

Figure 6.12: Database Recovery Attack for Foreign-key Join Queries for the TPC-H Benchmark.

for RANGE-SRC-SE-(α, x), since varying α does not affect the effectiveness of the attack.

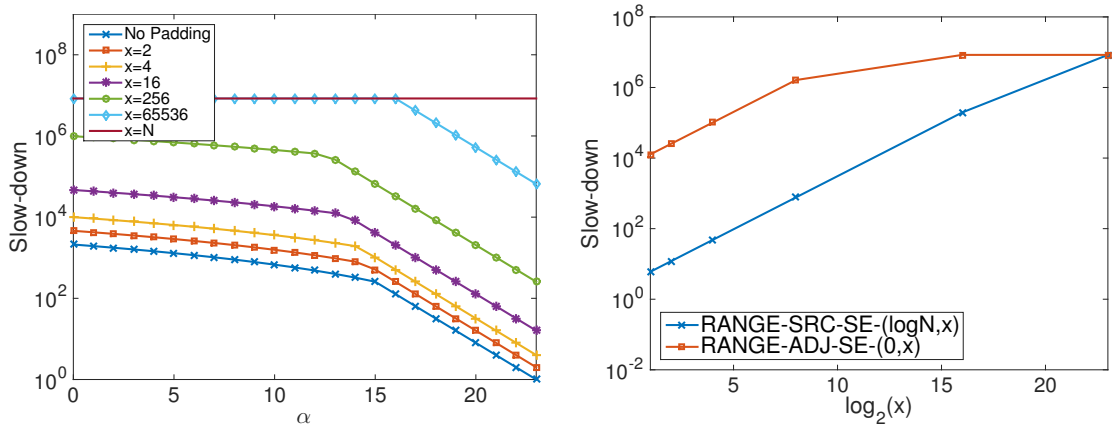
Attack evaluation. We focus on numeric attributes PS_SupplyCost from table PARTSUPP; P_Size and P_RetailPrice from PART; L_TAX, L_QUANTITY, L_DISCOUNT from LINEITEM. Table 6.1 presents for each attribute the number of all possible range queries and the number of the correctly decrypted ones using the baseline (Column 3 of Table 6.1), and RANGE-SRC-SE for $x = 2$, $x = 4$ and $x = 8$ (Columns 4, 5, 6 of Table 6.1). We observe that $x = 8$ drastically reduces the number of correctly decrypted queries. We omit the presentation of the database recovery attacks for ranges, since DR_{SR} is primarily based on the result of the query recovery attack, and we see in Table 6.1 that even for $x = 2$ QR_{SR} is small.

Attribute	#Queries	# Correctly Decrypted Queries			
		Baseline	RANGE-SRC-SE		
			$x = 2$	$x = 4$	$x = 8$
PS_SupplyCost	500500	73446	14	6	2
P_Size	1275	1184	10	5	2
P_RetailPrice	519690	19555	18	5	2
L_Tax	45	45	8	5	3
L_Quantity	1275	1263	10	4	3
L_Discount	66	66	8	4	1

Table 6.1: Query Recovery Attack for Range Queries QR_{SR} = # Correctly Decrypted Queries /# Queries)

6.4.6 Efficiency of Adjustable Constructions

In Figure 6.13(a), we fix a database with size 2^{22} records, and we show the largest slowdown (across all the possible result sizes—$1, 2, 3 \ldots N$) of SEAL(α, x) compared to a SE scheme which has the maximum leakage. Similarly, in Figure 6.14(a), we show the smallest speedup achieved by our construction SEAL(α, x) (for various values of α and x) compared to an approach that performs sequential scan and has no leakage. Because, we consider the worst-case speedup from the most secure solution ($\alpha = 0$ and $x = N$), sequential scan provides a more efficient approach than the use of worst-case padding with ORAM which is also achieves the same security. We do an analysis of these plots in the next section. We highlight again that **neither** SE **nor** sequential scan are competitors of SEAL, since (i) SEAL encapsulates those schemes (e.g., for $\alpha = 0$ and $x = N$ becomes sequential scan and for $\alpha = \log N$ and $x = \perp$ becomes SE scheme), and (ii) for non-trivial α and x they provide different security level. We provide those experiments only as reference points of SEAL's performance compared with the most and least se-



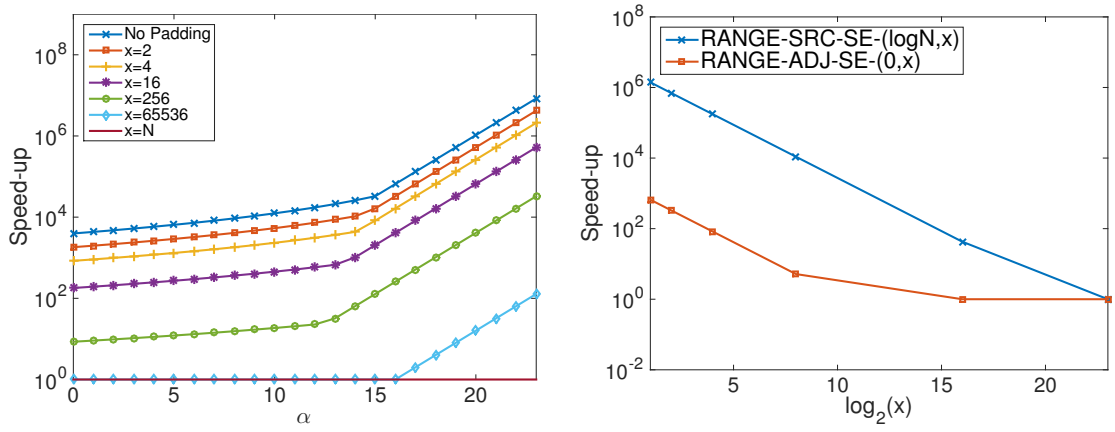
(a) SEAL(α, x) (b) Range Schemes

Figure 6.13: Slowdown from SE.

cure solutions. In addition, Figures 6.13(a),6.14(a) can be used in combination with Figures 6.8-6.12 and Table 6.1: For a given query type and attack we can specify good values for α, x (for mitigating the attack) from Figures 6.8-6.12 and Table 6.1,and for those values we can see the relative performance of SEAL compared with SE and sequential scan in Figures 6.13(a),6.14(a). Figure 6.13(b) and 6.14(b) evaluate RANGE-ADJ-SE-$(0, x)$ and RANGE-SRC-SE-$(\log N, x)$. Note that both schemes hide the overlapping pattern, the first by using ORAM, the second by construction. Also both schemes are using the same x, allowing the adversary to observe the same number of different sizes (but not necessarily the same sizes). Note that RANGE-SRC-SE performs much better than RANGE-ADJ-SE. This is to be expected given RANGE-SRC-SE has more leakage—the search pattern, which however we do not know how to use in an attack here.[4]

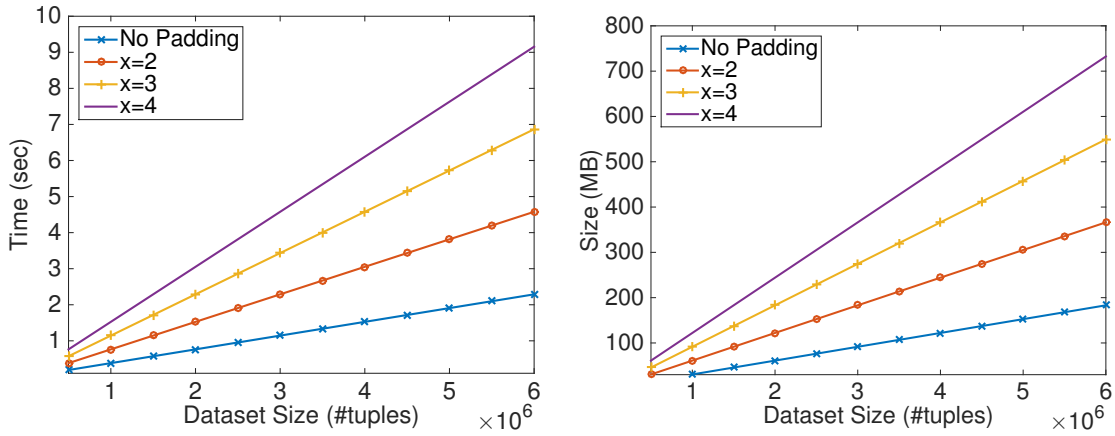
We provide additional experiments regarding the performance of our SEAL

[4]Although the search pattern (combined with the access pattern) has been used in recent work by Kornaropoulos et al. [84] to attack RANGE-SE, it is not clear how it can be used for RANGE-SRC-SE-(α, x).



(a) SEAL(α, x) (b) Range Schemes

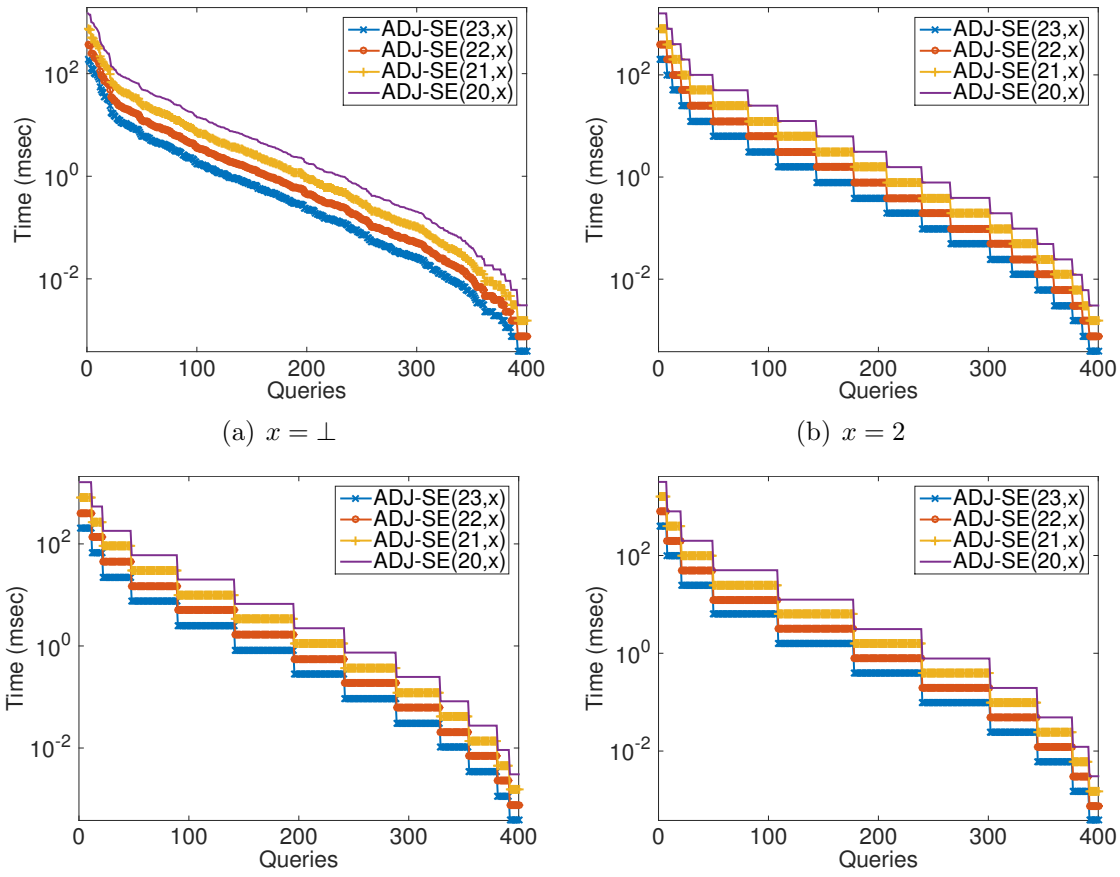
Figure 6.14: Speedup from sequential scan.



(a) Construction Time (b) Index Size

Figure 6.15: Index Costs - Crime Dataset

scheme for the crime dataset. We show experiments for values of α and x that significantly mitigate the proposed attacks and achieve good performance (as we also discuss in the next section). In Figure 6.15, we evaluate the required index size and construction time of SEAL for $x = \perp, 2, 3, 4$. Finally, in Figures 6.16 and 6.17 we evaluate the end-to-end search time of our SEAL scheme for two attributes of the crime dataset for $\alpha = 20, 21, 22, 23$ and $x = \perp, 2, 3, 4$.



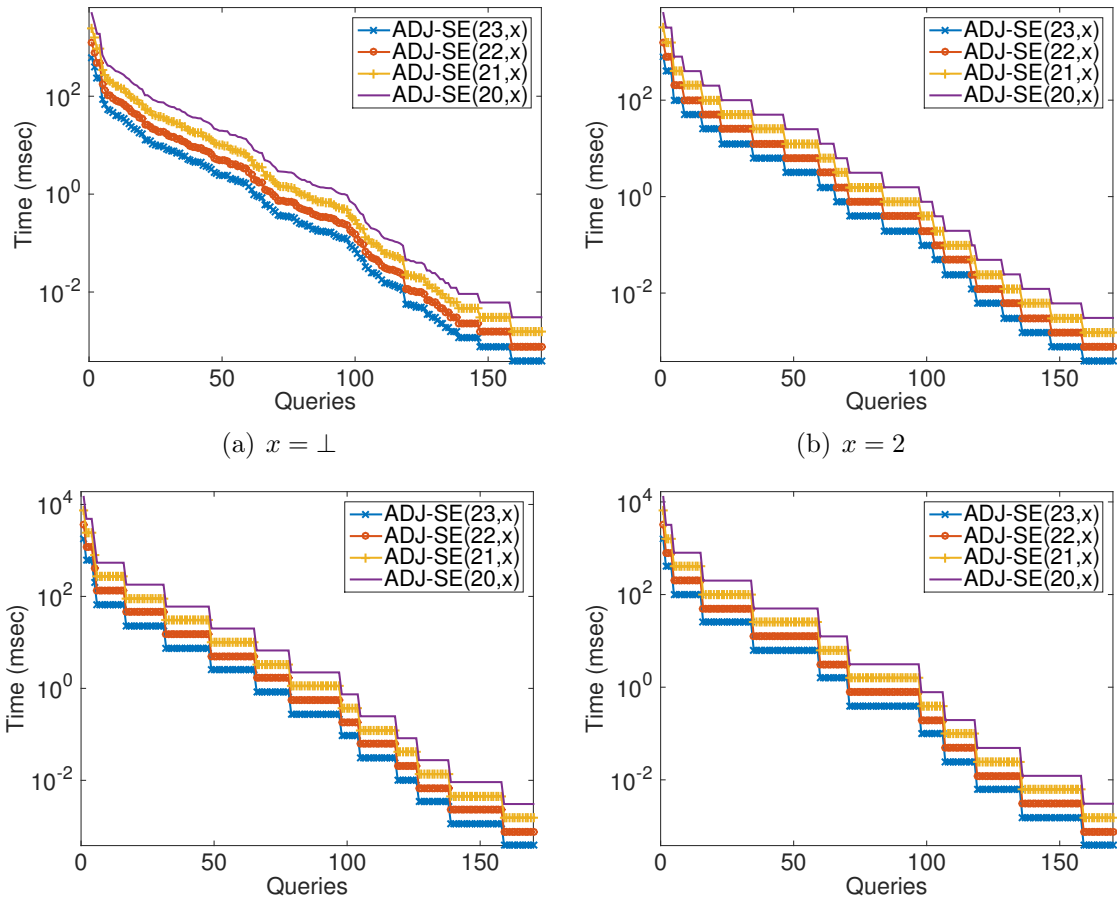
(a) $x = \perp$ (b) $x = 2$

(c) $x = 3$ (d) $x = 4$

Figure 6.16: Search costs - Crime Dataset (Attribute 5)

6.4.7 Setting Parameters α and x in Practice

From the above findings, it should be evident that finding appropriate parameter values is heavily data-dependent. In particular, it depends on the size of the database, number of distinct values, and the distribution of a given searchable attribute. One way for users to tune these parameters is to use our attacks as an estimator, e.g., provide their databases as input and try different values of α and x in order to set their desirable success rate thresholds against our attacks (before outsourcing the database). Below, we provide more general guidelines on how one



(a) $x = \perp$ (b) $x = 2$

(c) $x = 3$ (d) $x = 4$

Figure 6.17: Search costs - Crime Dataset (Attribute 8)

can set these parameters based on our evaluation.

Setting parameter x. Parameter x solely controls the success rate of the query recovery attack for point, range (RANGE-SRC scheme) and group-by queries. The query recovery attack tries to map the encrypted queries to plaintext ones based on the volume leakage. For instance, if a database contains only two values a and b and the volume of the former value is greater than the latter, i.e., $|q(a)| > |q(b)|$, the adversary can correctly map with certainty the encrypted query with the greater volume to a and the other one to b. Now, assuming that both values have the same volume, the adversary cannot distinguish the encrypted queries and is forced

to guess. Increasing the parameter x, we try to have more queries with the same size in order to increase the adversary's uncertainty, but finding a good value of x also depends on the distribution of the searchable value. For instance, attribute 9 of the crime dataset is a binary attribute (it has 2 distinct values), in which $|q(a)| = 4374175$ and $|q(b)| = 1749100$. We observe that for $x = 2$ these queries still will have different volumes, but for $x = 3$ they obtain the same volume (i.e., $|q'(a)| = |q'(b)| = 4782969$) and they will be indistinguishable. Attribute 10 of the crime dataset, which is also a binary attribute, has $|q(a)| = 5337429$ and $|q(b)| = 785846$ and in order to make these sizes indistinguishable higher values of x are needed, i.e., $x = 14$. Again, this kind of analysis can be performed locally, prior to outsourcing the dataset.

Setting parameter α. Parameter α affects the success rate of the database recovery attacks for point, range (RANGE-SRC scheme), join and group-by queries. The success of this attack firstly depends on the outcome of the query recovery attack. Thus, tuning the parameter x in order to increase the uncertainty of the adversary is very important. Nevertheless, parameter α controls how many tuples are indistinguishable from each other. For example, setting $\alpha = \log N - 1$ our scheme creates $N/2$ ORAMs of size 2—thus every tuple is indistinguishable from another one (all the tuples that are in the same ORAM are indistinguishable from each other). Therefore, even if the query recovery attack has 100% success rate and we are trying to find the correct mapping of plaintext tuples to encrypted ones, the success rate of this attack will be at most 50% for $\alpha = \log N - 1$. However, in our proposed database recovery attack, we treat the case when encrypted and plaintext

tuples have the same searchable value but differ in the rest of the attributes as a success. Due to this, the distribution of an attribute will also affect the success of the database recovery attacks. For instance, for point queries attribute 9 of the crime dataset (which has 2 values— $|q(a)| = 4374175$ and $|q(b)| = 1749100$) for $x = \perp$ and $\alpha = \log N - 1 = 22$, our attack has success rate around 87%, because the success rate of the query recovery attack is 100% and the adversary has uncertainty only when the same ORAM contains both tuples with value a and b.

Finally, we provide some general conclusions from the analysis that we performed on our chosen datasets. We observe that for point and join queries setting $\alpha = \log N - 3$ and $x = 4$ significantly reduces both QR_{SR} and DR_{SR} (e.g., attributes 4,5 of LINEITEM and attributes 13,14 of crime dataset for point queries; SUPPLIER⋈NATION and CUSTOMER⋈NATION for join queries), while for these values the smallest speedup from sequential scan is more than $262,000\times$ and the maximum slow-down from SE is $32\times$. There are rare cases that attributes with skewed distribution and small number of distinct values, e.g., binary attributes, require higher values of x, such as $x = 16$ or $x = 64$ (e.g., attribute 9 of LINEITEM and attributes 9,10 of the crime dataset for point queries). In the cases of range queries, we observe that our RANGE-SRC-SE-$(\log N, x)$ for $x = 8$ significantly mitigates our all-powerful query recovery attack (e.g., L_Tax and L_Discount attribute—the success rate of the attack drops from 100% below 7% and 2% respectively) and achieves a maximum $48\times$ slowdown from plain RANGE-SE.

6.5 Challenges for Dynamic Databases

Our work only focuses on static databases. We believe that a very interesting problem for future work is to extend this work for dynamic databases, an approach that introduces more leakage and makes the problem more challenging. Towards this goal, we know from the literature of SE how we can support dynamic point queries (there is an extensive literature on dynamic schemes that achieve forward/backward privacy[24, 36, 65, 66, 67, 72]—the state-of-the-art security definitions for dynamic SE. A first challenge towards dynamic databases is to study if these security definitions for point queries are suitable for other query types (such as range, joins and group-by queries), as well as to find schemes that achieve those definitions. A second challenge is that prior ORAM and our ADJ-ORAM schemes require initializing at setup the worst-case memory size—modifying them for the dynamic case (without having to set a-priori a large upper bound) is a non-trivial problem. A third challenge is how we could efficiently use our ADJ-Padding technique, since new updates will continuously change the distribution of the searchable attribute. Predicting the required padding size (without extra costly bookkeeping) for a certain keyword without knowing future updates would be very challenging.

One approach for handling dynamic point queries would be to explore whether our ADJ-ORAM can be used as a drop-in replacement in existing dynamic ORAM-based SE schemes (e.g., ORION from [66]), obtaining a good efficiency/security trade-off. However, this would require addressing the aforementioned second and third challenges. An alternative direction that avoids these challenges is to use ex-

isting techniques that transform static SE to dynamic ones (e.g., SD_a from [36]). At a high level, this requires storing the result of N updates in a sequence of $\log N + 1$ separate indexes (with size $2^0, \ldots, 2^{\log N}$), where each update is first stored in the smallest index and whenever two indexes of the same size exist they are downloaded and merged to a larger new index by the client. Search queries are executed at all encrypted indexes independently. Such techniques that periodically rebuild the encrypted indexes do not require defining a maximum capacity during setup. Moreover, they allow the client to update the parameters α and x depending on how the database has evolved. However, the main drawback of this approach is updates, since it has a (amortized) $O(\log N)$ update cost. While de-amortization is possible, it is not trivial, especially in our adjustable setting, and we believe that it is a very interesting problem for future work.

Chapter 7: Conclusions and Future Work

7.1 Conclusions

In this thesis, we focus on building cryptographic solutions for encrypted search and computation that are simultaneously practical and provable secure.

Towards this goal, we propose novel, **more efficient** SE schemes with better search/computation time that improve in practice the locality of SE schemes and reduce the number of required cryptographic operations during search. Combining together these approaches can lead to search time improvements of up to 3-4 orders of magnitude compared with prior state-of-the-art SE schemes.

Furthermore, this thesis also focuses on extending SE to support new, more expressive private queries in an efficient and secure manner. In particular, we propose novel forward-and-backward private DSE schemes with small client storage, which outperform prior state-of-the-art schemes by up to 4 orders of magnitude. We also propose the state-of-the-art private range scheme, which is efficient and robust against not only previously proposed access/volume attacks, but also new very powerful attacks (in which the attacker has access to the input plaintext database).

Finally, we present SEAL, a family of new, **more secure** SE schemes with adjustable leakages that can be used to support basic database queries, such as

point, range, join and group-by queries. In SEAL, the amount of privacy loss is expressed in leaked bits of search or access pattern and can be defined at setup. As our experiments show, when protecting only a few bits of leakage (e.g., three to four bits of access pattern), enough for existing and even new more aggressive attacks to fail, the query execution time of SEAL is within the realm of practical for real-world applications. Our findings show that SEAL could be a promising approach for building efficient and robust encrypted databases.

7.2 Future Work

Next Generation Encrypted Databases. There is a long line of research on privacy preserving databases that attempts to strike a desirable balance between security and practical efficiency. Previous approaches, especially encountered in database venues, provide practical solutions, while offering a wide range of different functionalities commonly offered in plaintext databases. However, these approaches either lack provable security guarantees, or permit unacceptable leakages. At the same time, there are other approaches primarily inspired by the crypto community that provide well-defined security guarantees, but their solutions are not practical for large-scale database applications. Bridging the gap between these two worlds to build real-world encrypted databases and encrypted systems that will simultaneously provide reasonable in practice security and efficiency guarantees, remains an open problem. Towards this goal, the last couple of years we observe new, more efficient, expressive and secure solutions for encrypted search, and we are ready to

start building the next generation of encrypted systems/databases. However, the challenges that arise from building such solutions are two-fold: (i) leveraging and further improving the existing solutions, (ii) addressing new security and efficiency research challenges that will appear when attempting to combine existing state-of-the-art schemes in a single system. For instance, it is an open problem to combine different private query operators and define the leakage of composite queries (e.g., a combination of join, range and group-by queries). At the same time, we observe the need for a new query optimizer that takes into consideration, not only the efficiency, but also the security of a query plan. Furthermore, we need to study the existence of new attacks for these systems and find a way to efficiently mitigate them, e.g., extending SEAL ([37]) for more complex queries.

Practical oblivious primitives. A standard way to transform a non-oblivious algorithm to an oblivious one is to perform all the memory accesses obliviously using an ORAM scheme. Thus, the question of *"how can we further improve the practicality of ORAM schemes"* arises. For instance, in [31, 37, 38], we proposed new, more efficient ORAM schemes that are locality-aware, provide an adjustable trade-off between security and efficiency, and comprise the state-of-the-art schemes for secure hardware. However, there are various other ways to further improve ORAM schemes, such as by designing: (i) efficient dynamic ORAM schemes—current state-of-the-art approaches require pre-allocating the maximum required memory during the setup process, (ii) new parallel and distributed practical ORAM schemes—current approaches are mainly of theoretical interest, (iii) new ORAM schemes for searchable encryption—current approaches are optimized for retrieving one memory location

at a time; however searchable encryption requires accessing results of variable sizes.

Towards a more practical searchable encryption. Since the first work by Song et al.[16] in 2000, various dimensions of SE have been studied, such as security, dynamism, better security for dynamic schemes, parallelism, locality, read efficiency, index size, number of cryptographic operations per query, and expressiveness. The existence of these dimensions paves the way to interesting research questions, such as how to further improve previous state-of-the-art results, or how to combine these different dimensions in order to further enhance the practicality of SE schemes. For instance, while in the last two years we have observed numerous works on dynamic and locality-aware SE schemes, there is limited knowledge on the impact of combining these dimensions. In addition, the majority of the efficiency advancements mainly focuses on the keyword search problem, and there are many unanswered questions regarding the impact of combining the efficiency/security dimensions with other query types. For instance, it remains an open problem to see if for more expressive query types, e.g., range, join, group-by, graph queries, we can extend current state-of-the-art static approaches to dynamic with forward and backward privacy guarantees. Finally, another important next step for the future of SE is to find other important dimensions for this problem—attempting to extend SE to a multi-user scenario will raise new security and efficiency challenges, such as providing individual privacy per user and designing new schemes with better concurrency.

Bibliography

[1] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, M Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*, 2014.

[2] **I. Demertzis**, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. Practical private range search revisited. In *SIGMOD*, 2016.

[3] **I. Demertzis**, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *TODS*, 2018.

[4] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In *CRYPTO*. 2010.

[5] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path Oram: An Extremely Simple Oblivious Ram Protocol. In *CCS*, 2013.

[6] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, 2017.

[7] Saba Eskandarian and Matei Zaharia. Oblidb: oblivious query processing for secure databases. *PVLDB*, 2019.

[8] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *SP*, 2018.

[9] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *TKDE*, 2013.

[10] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *SP*, 2015.

[11] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *SOSP*, 2011.

[12] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

[13] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[14] Sumeet Bajaj and Radu Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 205–216. ACM, 2011.

[15] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*, 2015.

[16] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *SP*, 2000.

[17] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[18] Eu-Jin Goh et al. Secure Indexes. *IACR Cryptology ePrint Archive*, 2003.

[19] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*, 2006.

[20] Melissa Chase and Seny Kamara. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*, 2010.

[21] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Computationally Efficient Searchable Symmetric Encryption. In *SDM*. 2010.

[22] Kaoru Kurosawa and Yasuhiro Ohtaki. UC-Secure Searchable Symmetric Encryption. In *FC, 2012*.

[23] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic Searchable Symmetric Encryption. In *CCS*, 2012.

[24] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS*, 2014.

[25] Seny Kamara and Charalampos Papamanthou. Parallel and Dynamic Searchable Symmetric Encryption. In *FC*, 2013.

[26] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*, 2013.

[27] Ian Miers and Payman Mohassel. IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality. In *NDSS*, 2017.

[28] Raphael Bost. Sofos: Forward Secure Searchable Encryption. In *CCS*, 2016.

[29] David Cash and Stefano Tessaro. The Locality of Searchable Symmetric Encryption. In *EUROCRYPT, 2014*.

[30] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable Symmetric Encryption: Optimal Locality in Linear Space via Two-Dimensional Balanced Allocations. In *STOC, 2016*.

[31] **I. Demertzis**, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. *CRYPTO*, 2018.

[32] **I. Demertzis** and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[33] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT*, 2017.

[34] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*, 2015.

[35] **I. Demertzis**, Rajdeep Talapatra, and Charalampos Papamanthou. Efficient searchable encryption through compression. *PVLDB*, 2018.

[36] **I. Demertzis**, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[37] **I. Demertzis**, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX*, 2020.

[38] Javad Ghareh Chamani, Dimitrios Papadopoulos, **I. Demertzis**, Charalampos Papamanthou, and Rasool Jalili. GraphOS: Towards oblivious graph processing. *(under submission)*.

[39] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *Journal of Computer Security, 2011*.

[40] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM, 1996*.

[41] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, 2014.

[42] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):129, 2013.

[43] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. *22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[44] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[45] Kesheng Wu, Ekow J Otoo, Arie Shoshani, and Henrik Nordberg. Notes on design and implementation of compressed bit vectors. Technical report, Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[46] Crimes 2001 to present (city of chicago). https://data.cityofchicago.org/ public-safety/crimes-2001-to-present/ijzp-q8t2.

[47] Enron email dataset. https://www.cs.cmu.edu/enron/.

[48] Usps, http://www.app.com/.

[49] Tpc-h benchmark. http://www.tpc.org/tpch.

[50] Gilad Asharov, Gil Segev, and Ido Shahaf. Tight tradeoff s in searchable symmetric encryption. CRYPTO, 2018.

[51] AH Team and Others. Apache HBase Reference Guide.

[52] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-store 7 Years Later. *PVLDB*, 2012.

[53] Bouncy castle. http://www.bouncycastle.org.

[54] James Kelley and Roberto Tamassia. Secure compression: Theory and practice. *IACR Cryptology ePrint Archive*, 2014, 2014.

[55] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. SP, 2018.

[56] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Oblix: An Efficient Oblivious Search Index*. SP, 2018.

[57] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, 2017.

[58] Doug Cutting and Jan Pedersen. Optimization for dynamic inverted index maintenance. In *SIGIR*, 1989.

[59] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: a dissection and experimental evaluation. *PVLDB*, 9(12):1113–1124, 2016.

[60] Lemire. Javafastpfor, Apr 2017.

[61] Lemire. javaewah, Dec 2016.

[62] M.f. Porter. An algorithm for suffix stripping. *Program*, 14(3):130137, 1980.

[63] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *ACNS*, 2005.

[64] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX 2016*.

[65] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[66] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[67] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PETS*, 2018.

[68] Seny Kamara and Tarik Moataz. Sql on structurally-encrypted databases. *ASIACRYPT*, 2019.

[69] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *SP*, 2016.

[70] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *CRYPTO*, 2016.

[71] Shi-Feng Sun, Xingliang Yuan, Joseph K Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *CCS*, 2018.

[72] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *CCS*, 2017.

[73] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inf. Process. Lett.*, 1981.

[74] OpenSSL: The open source toolkit for SSL/TLS. `https://www.openssl.org/`, 2003.

[75] Javad Ghareh Chamani. Implementation of Mitra, Orion, Horus, Fides, and Diana_Del. `https://github.com/jgharehchamani/SSE`, 2018.

[76] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 235–246. ACM, 2014.

[77] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[78] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS*, 2020.

[79] Jonathan L Dautrich Jr and Chinya V Ravishankar. Compromising Privacy in Precise Query Protocols. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 155–166. ACM, 2013.

[80] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[81] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*, 2018.

[82] Paul Grubbs, Marie-Sarah Lacharit, Brice Minaud, and Kenny Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range series. In *CCS*, 2018.

[83] Evangelia Anna Markatou and Roberto Tamassia. Full database reconstruction with access and search pattern leakage. ISC 2019, 2019.

[84] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. IEEE SSP 2020, 2020.

[85] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *SP*, 2019.

[86] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *CCS*, 2019.

[87] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *SP*, 2019.

[88] Attack of the week: searchable encryption and the ever-expanding leakage function. `https://blog.cryptographyengineering.com/`. Accessed: 2019-06-06.

[89] Seny Kamara and Tarik Moataz. Encrypted multi-maps with computationally-secure leakage. 2019.

[90] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious ram. *Proceedings on Privacy Enhancing Technologies*, 2018.

[91] Raphael Bost and Pierre-Alain Fouque. Thwarting leakage abuse attacks against searchable encryption–a formal approach and applications to database padding. Technical report, Cryptology ePrint Archive, Report 2017/1060.

[92] Marie-Sarah Lacharité and Kenneth G Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018.

[93] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *CCS*, 2006.

[94] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[95] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 162–168, 2017.

[96] Louis Granboulan and Thomas Pornin. Perfect block ciphers with small blocks. In *International Workshop on FSE*, 2007.

[97] Emil Stefanov and Elaine Shi. Fastprp: Fast pseudo-random permutations for small domains. *IACR*, 2012.

[98] Ben Morris and Phillip Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *EUROCRYPT*, 2014.

[99] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *FOCS*, 2018.