Secure and Efficient Searchable Encryption with Leakage Protection

Shujie Cui

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science, The University of Auckland, 2019.

To all those who contributed to the work described in this thesis, and to all those who will read this in the future and find it useful.

Acknowledgements

The work presented in this thesis would not have been possible without the collaboration of a considerable number of people to whom I would like to express my gratitude.

First and foremost, I would like to deeply thank my supervisors, Giovanni Russello, Steven D. Galbraith, and Muhammad Rizwan Asghar, for providing all the conditions and the environment for exploring new ideas, and for helping me overcome all challenges I had to face during my Ph.D. research. Whenever I encountered problems and difficulties, they always gave me suggestions and ideas on how to proceed. Without their encouragement, patient guidance, and endless efforts in helping me, I could not finish this thesis.

Second, my sincere gratitude further goes to my colleagues Ming Zhang, Rong Yang, and Jin Yang for helping me implement the prototypes of the systems presented in this thesis. Without their help, I could not have evaluated my proposals precisely and efficiently. I am also thankful for my other labmates at the University of Auckland, including Sana Belguith, Lukas Zobernig, and Zhijie Li. Many thanks to them for being so supportive and helpful during my Ph.D. study and for making our lab a lovely place to work.

Third, this research is supported by STRATUS (Security Technologies Returning Accountability, Trust and User-Centric Services in the Cloud), a project funded by the Ministry of Business, Innovation and Employment (MBIE), New Zealand. I am grateful to acknowledge its support.

Last but not least, I am grateful to my parents, my grandparents, and my brother, for their love and encouragement. I dedicate this thesis to them!

Abstract

Cloud computing is a successful paradigm offering companies and individuals unlimited data storage and computational power at attractive costs. Despite its benefits, cloud computing raises security concerns for sensitive data. Once the data is outsourced, it is directly exposed to careless or potentially malicious Cloud Service Providers (CSPs). Moreover, the data can be learned by intruders due to the possible compromise of the cloud platform. To protect the outsourced data, it is necessary to encrypt the data before uploading them to the CSP. However, standard cryptographic primitives do not allow the CSP to do any operation over the encrypted data, including search.

The concept of Searchable Encryption (SE) provides a more promising solution to support searching over encrypted data while protecting outsourced data from unauthorised accesses by CSPs. In the literature, a plethora of SE schemes have been proposed. Unfortunately, a majority of them expose information about data and queries, called *leakage*, to the CSP. In recent years, a series of works illustrate that the CSP is potentially able to recover the content of data by analysing the leakage.

This thesis presents three different SE schemes with minimised leakage, which not only ensure the confidentiality of the data, but also resist existing leakage-based attacks. Moreover, in multi-user settings, the proposed schemes do not need to update the key or re-encrypt the data when revoking compromised users. Meanwhile, the proposed SE schemes guarantee a practical user experience, where users only need to encrypt queries and decrypt results. The first proposed SE scheme is built on top of the hybrid cloud infrastructure, where a trusted private cloud platform is deployed between users and the public CSP. The second solution is designed for the organisations without private cloud platforms, which can be deployed across two public CSPs. The last solution combines SE with Intel Software Guard Extension (SGX), a trusted hardware that can be embedded in the CSP. We present a theoretical security analysis for the three schemes. We also implemented prototypes and evaluated their performance. The results demonstrate that the proposed schemes can efficiently address the challenging problems in practice.

Contents

Acknowledgements iii

Abstract v

Contents viii

List of Figures xii

List of Tables xiii

List of Algorithms xiv

Nomenclature xiv

1 Introduction 1

1.1 Motivation 1
1.2 Design Goals 4
1.3 System Settings 4
1.4 Our Contributions 5
1.5 Organisation 7

2 Leakage and Attacks 9

2.1 Notations 9
2.2 Leakage Level Definition 12
2.3 Attacks against SE Solutions 15
2.3.1 Frequency Analysis Attack 15
2.3.2 IKK Attack 16
2.3.3 File-injection and Record-injection Attack 17
2.3.4 Count and Relational-count Attack 17
2.3.5 Reconstruction Attack 18

2.4 Countermeasures and Challenges . . . 19
2.4.1 Size Information . . . 19
2.4.2 Search Pattern . . . 20
2.4.3 Access Pattern . . . 21
2.4.4 Forward and Backward Privacy . . . 23

3 Dummy Records Generation **25**
3.1 Security Requirements . . . 25
3.2 Static Databases . . . 27
3.3 Dynamic Databases . . . 31
3.3.1 Solution 1 . . . 31
3.3.2 Solution 2 . . . 34
3.3.3 Solution 3 . . . 36
3.3.4 Conclusion . . . 37

4 Hybrid Cloud Based Solution: ObliviousDB **39**
4.1 Overview of *ObliviousDB* . . . 41
4.1.1 System Model . . . 41
4.1.2 Proposed Approach . . . 42
4.2 Solution Details . . . 44
4.2.1 Setup . . . 44
4.2.2 Group Generation . . . 44
4.2.3 Data Encryption . . . 46
4.2.4 Query Encryption and Execution . . . 48
4.2.5 Oblivious Algorithm . . . 50
4.2.6 Database Updating . . . 51
4.3 Security Analysis . . . 53
4.4 Performance Analysis . . . 57
4.4.1 Group Generation . . . 58
4.4.2 Select Query Evaluation . . . 59
4.4.3 Insert Query Evaluation . . . 61
4.5 Related Work . . . 62
4.5.1 Schemes with Controlled Leakage . . . 62
4.5.2 Multi-user SE Schemes . . . 63
4.6 Conclusions and Future Work . . . 64

5 Multi-Cloud Based Solution **67**
5.1 Solution Overview . . . 69
5.1.1 System Model . . . 69
5.1.2 Threat Model . . . 70
5.1.3 Approach Overview . . . 70
5.2 Solution details . . . 71
5.2.1 Setup . . . 71
5.2.2 Select Query . . . 74
5.2.3 Shuffling and Re-randomisation . . . 77
5.2.4 User Revocation . . . 78
5.2.5 Database Updating . . . 78
5.3 Security Analysis . . . 80
5.3.1 The Security Analysis against the SSS . . . 81
5.3.2 The Security Analysis against the WSS . . . 82
5.4 Performance Analysis . . . 84
5.4.1 Complexity Analysis of *P-McDb* . . . 84
5.4.2 Benchmark of *P-McDb* . . . 86
5.5 Discussion . . . 90
5.6 Related SE schemes Using multiple Servers . . . 90
5.7 Conclusion . . . 91

6 Preserving Access Pattern Privacy in SGX-Assisted Encrypted Search **93**
6.1 Background . . . 96
6.1.1 Intel SGX . . . 96
6.1.2 Side Channel Attacks on SGX . . . 96
6.2 Solution Overview . . . 97
6.2.1 System Entities . . . 97
6.2.2 Threat Model and Assumptions . . . 98
6.2.3 Architecture Overview . . . 98
6.2.4 Security Definition . . . 99
6.3 Solution Detail . . . 99
6.3.1 Data Representation . . . 99
6.3.2 Data Encryption . . . 102
6.3.3 Searching Tree . . . 102
6.3.4 Equality Query . . . 105
6.3.5 Range Query . . . 106
6.3.6 Returning Search Results . . . 108

6.4 Complex Queries . . . 108
6.4.1 Aggregate Functions . . . 109
6.4.2 Join Queries . . . 110
6.5 Security Analysis . . . 112
6.6 Performance Analysis . . . 114
6.6.1 Implementation . . . 115
6.6.2 TPC-H Benchmarking . . . 115
6.6.3 Big Data Benchmarking . . . 119
6.7 Related Work . . . 119
6.8 Conclusions and Future Work . . . 120

7 Conclusions and Future Work **123**
7.1 Summary . . . 123
7.2 Future Work . . . 124

References **127**

List of My Publications **141**

List of Figures

3.1 The number of required dummy records with different group sizes for TPC-H 'ORDERS' dataset. 27

4.1 An overview of *ObliviousDB*. 41
4.2 The process time taken by each phase when changing the number of groups. . . 59
4.3 The process time taken by each entity when changing the number of groups. . . 59
4.4 The process time taken by each phase when changing the result size. 60
4.5 The process time taken by each entity when changing the result size. 60
4.6 The performance of insert queries. 61

5.1 An overview of *P-McDb*. 69
5.2 The performance of the WSS in different cases. 86
5.3 The performance of each operation and entity. 87
5.4 The performance of the insert query. 89

6.1 An overview of the proposed approach. 97
6.2 A B+ tree index example with $branch = 4$, 3 levels, and 13 nodes. 100
6.3 Tree search time with 1 million keys. The branch factoring=32. 115
6.4 The time of finding the MAX value among 1000 matched records. 117
6.5 The time of executing a join query. 118

List of Tables

2.1 A comparison of SE schemes. 11
2.2 Summary of leakage profiles and attacks against encrypted databases. 15

3.1 Comparison of the solutions for managing dummy records. 37

4.1 Data representation in *ObliviousDB*. 43
4.2 The storage overhead with different numbers of groups. 58

5.1 Data representation in *P-McDb*. 72
5.2 The computation and communication overhead for each record or each query. . 85

6.1 Comparison of recent SGX-based schemes. 94
6.2 The details of the tested queries. 115
6.3 Comparison with ObliDB and the baseline. 116

List of Algorithms

1 $RcdEnc(rcd, flag, s)$ . . . 46
2 $NonceBlind(Ercd, Grcd, GDB, counter)$ . . . 47
3 $Query(Q, s)$ . . . 48
4 $Oblivious(EQ.f, GE(Q.e))$ . . . 50
5 $Insert(rcd, s)$ . . . 51
6 $RcdEnc(rcd, flag, s_1, s_2)$ . . . 73
7 Query(Q, s_1, s_2) . . . 75
8 $Shuffle(IL, s_2)$ . . . 77
9 $Insert(rcd)$ . . . 79
10 Query$(EQ, tree, P)$ . . . 103
11 EqualitySearch$(Q, Nodes, b, mid, isLastLevel)$ . . . 105
12 RangeSearch$(Q, Nodes, b, mid)$ . . . 107
13 MAX$(Ercds, rids, m)$ . . . 108
14 SUM$(Ercds, rids, m)$ . . . 109
15 Join$(tree_1, tree_2)$ . . . 110
16 JoinSearch$(Nodes_1, len_1, Nodes_2, len_2, res)$ . . . 110

Chapter 1

Introduction

1.1 Motivation

Cloud computing is a successful paradigm offering companies and individuals virtually unlimited data storage and computational power at very attractive costs. Despite its benefits, cloud computing raises serious security concerns for preserving the confidentiality of sensitive data, such as medical, social, and financial information. Once the data is outsourced, it is exposed to careless or even potentially malicious Cloud Service Providers (CSPs). Moreover, the data could also be learned by third party intruders because the cloud platform could be compromised. In this context, the data owner lacks a valid mechanism for protecting the data from unauthorised access.

A straightforward method to solve the issue is to encrypt the data with standard cryptographic primitives, such as AES and 3DES, before uploading it to the CSP. However, this method is not practical for the applications requiring to perform search over the data, such as relational databases, web applications, and machine learning tools. The reason is that standard cryptographic primitives do not support search operations over encrypted data. If a piece of data is required, a trivial solution is that the user downloads all the content to its local (trusted) environment, decrypts the data, and performs the search operation. Unfortunately, this trivial solution does not scale well when the database is very large. Alternatively, we can provide the CSP with the key to decrypt all data. Then, the CSP can search the decrypted data to retrieve the required part. This method is used in most of the commercial public cloud services, such as Amazon S3 [1]. However, this approach still allows the CSP to learn the content of data and queries.

In privacy-sensitive applications, both the data and the query imply privacy information of data owners and users. For instance, in electronic healthcare systems, medical records contain physical conditions of patients, and the queries issued by doctors indicate their specialities and

medical preferences. In practice, the CSP might be trusted. However, they can be compromised and controlled by external adversaries. Moreover, an employee in the CSP company could also be malicious. The attacker who compromises the cloud server or the malicious employee in the CSP might sell such information to insurance and medical companies for personal benefits. For simplicity, in the rest of this work, we treat the CSP as the attacker who makes effort to learn the content of data and queries and infer user privacy. Therefore, it is important to ensure the confidentiality of both of the data and queries when outsourcing sensitive data to the cloud.

The concept of Searchable Encryption (SE) provides a promising solution to support searching over encrypted data while protecting outsourced data from unauthorised accesses by CSPs. In SE schemes, encrypted data is tagged with encrypted keywords in such a way that a CSP, which is given an encrypted search term, can check whether a record has the keywords that satisfy the search term without decrypting the record or the search term. In other words, SE not only supports search operations but also ensures confidentiality of both the record and the query. Based on the underlying encryption primitives, SE schemes could be Symmetric Searchable Encryption (SSE) or Public key Encryption with Keyword Search (PEKS) [2]. In general, SSE is much more efficient than PEKS, yet PEKS is more flexible to support multi-user access.

Since the seminal paper by Song *et al.* [3], a plethora of SE schemes have been proposed. A long line of works, such as [3–24], focus on investigating SE with complex functionality (*e.g.,* multi-keyword search, range queries, rank search, boolean search, and fuzzy search) and improved performance. Unfortunately, a majority of them expose sensitive information about data and queries, called *leakage*, to the CSP. For instance, from the encrypted database, the CSP might learn the number of times each keyword appears in the database (we call this *frequency information*). From the search operations, the CSP is able to see the *access pattern*, *i.e.,* the physical location and ciphertext of encrypted data accessed by a given query. Moreover, the CSP can also infer if two or more queries have the same underlying keywords or not (if the queries are equivalent or not, or if the queries search for the same data or not), referred to as the *search pattern*, by comparing the encrypted queries or their matched data. Last but not least, the CSP can simply log the number of matched records or files returned by each query, referred to as the *size pattern*.

A series of more recent works [25–30] illustrate that the CSP is potentially able to recover the content of queries and data by analysing the leakage. For instance, Islam, Kuzu and Kantarcioglu [25] showed how the access and search patterns can be exploited to recover queries. This attack is referred to as *IKK attack* in the literature. Naveed *et al.* [26] recover more than 60% of the data stored in CryptDB [17] using frequency analysis. Cash *et al.* [27] give a comprehensive analysis of the leakage in SE solutions for file collections and introduced the *count attack*, where the CSP could recover queries by counting the number of matched records even if the encrypted records are semantically secure. Unfortunately, the majority of the existing SE

solutions, such as [3–24], are vulnerable to these attacks due to the leakage of search, access, and size patterns.

In addition to the search, access, and size patterns, the SE scheme might leak more information when executing insert and delete queries. Based on whether the scheme still supports data updating after the database bootstrapping, the SE schemes are generally categorised into *static* and *dynamic*. In *static SE* schemes, such as [8, 21, 31], users cannot update the data once the database is encrypted and uploaded to the CSP. In contrast, *dynamic SE* schemes, *e.g.,* [6, 14, 19], still support insert, update, and delete operations after bootstrapping the database. Dynamic SE schemes leak extra information if they do not support *forward privacy* and *backward privacy* properties. Lacking forward privacy means that the CSP can learn if newly inserted data or updated data matches previously executed queries. Lacking backward privacy means that the CSP learns if deleted data matches new queries. In [28], Zhang *et al.* investigate the consequences when dynamic SE schemes fail to ensure the forward privacy, and introduce the *file-injection attack*. Specifically, the CSP could recover a very high fraction of searched keywords by injecting a small number of known files and observing the access pattern. Supporting forward and backward privacy is fundamental to limit the power of the CSP to collect information on how the data evolves over time. Several of the existing schemes, such as [7, 19, 32, 33], support forward privacy, but only [34, 35] are able to support both properties simultaneously.

Privacy-sensitive applications, such as electronic healthcare systems, impose stringent security requirements when it comes to data outsourcing. The first step to meet those requirements is to minimise information leakage that could lead to *inference attacks*, *i.e.,* the attacks where the CSP can infer the content of the data or the queries based on the obtained leakage. Oblivious Random Access Memory (ORAM) [36–38] is a cryptographic primitive that allows a client to access memory locations from a server without revealing the access pattern. Homomorphic Encryption (HE) [39, 40] is a form of encryption supporting computation on encrypted data. Moreover, after decryption, the computation result is the same as the result of performing identical operations on the plaintext. As shown in [41–43], both the ORAM and HE techniques can be leveraged in SE schemes to protect the patterns from the CSP. However, existing ORAM and HE techniques are prohibitively costly and impractical. In this thesis, we are concerned with practical solutions where users can get the required data efficiently without incurring impractical storage, communication and computation overheads. Moreover, in multi-user settings, where the systems allow multiple users to read and/or write the data, users could join or leave the system at any time, ideally without affecting the rest of the users. Unfortunately, ORAM-based and HE-based solutions fail to cover these important aspects. Therefore, a fundamental problem is to develop secure and practical SE schemes for privacy-sensitive applications in multi-user settings.

1.2 Design Goals

Based on the above discussion, the goals of our SE schemes can be summarised below:

- First, we ensure confidentiality of the records and queries while the CSP is still able to return the encrypted result to users correctly.
- Second, we aim at protecting the search, access, and size patterns from the CSP, and ensuring the forward and backward privacy of the database, such that the CSP cannot mount any leakage-based inference attack.
- Third, we also consider the key management for multi-user databases. Specifically, our system allows revoking compromised users efficiently without regenerating the secret key and re-encrypting the records.
- Fourth, we aim at achieving a practical experience on the user side. Specifically, the user should be able to get the query result in an acceptable time. Moreover, the storage, communication and computation overhead on the user side should be reasonable.

1.3 System Settings

Before discussing the contributions, we describe the system settings in our work.

In this thesis, we propose several incremental approaches to protect the confidentiality of the data stored in outsourced databases. In particular, we focus on protecting the records stored in relational databases. Roughly speaking, a relational database is a set of tables organised into *rows* and *columns*. Each row in a table, also called a *record* or a *tuple*, consists of a sequence of elements and represents an entity or item. Each column, or *field*, represents an attribute of the entity. All the data referenced by a field is in the same domain and conform to the same constraints. One single *data value* or *element* represents a cell in the table. Herein, we refer to the *database* as a single table.

As mentioned above, based on whether the encrypted database supports insert, update and delete queries, we categorise the encrypted databases into *static* and *dynamic*. A static database, such as country names all over the world, only supports select queries. On the contrary, in dynamic encrypted databases, *e.g.,* electronic healthcare systems, the user can still insert, delete, or update records. In this thesis, we will discuss how to minimise information leakage for both static and dynamic databases.

Moreover, we further categorise the dynamic databases according to their access settings into Single-User (SU), Single-writer Multi-reader (SwMr) and Multi-writer Multi-reader (MwMr) scenarios. Basically, in *single-user* databases, only the data owner can issue queries to read and

write the records. *SwMr* databases allow the data owner to insert, update, and delete records, whereas, the other users are only authorised to issue select queries for data reading. In *MwMr* schemes, multiple users can read and write the records. In all scenarios, the data owner aims at protecting its data from the CSP and unauthorised users. Moreover, the data user aims at concealing its queries and search results from others, including the CSP, the data owner, and other users. The work in this thesis presents approaches to manage secret keys, and methods to minimise information leakage for both the SwMr and MwMr applications.

Note that the static database is uploaded during the bootstrapping by the data owner or the system administrator, and none can update it after the bootstrapping. Therefore, the static database only supports single-user and SwMr scenarios.

1.4 Our Contributions

To achieve the above goals, we first summarise the leakage in existing SE schemes and analyse how the leakage is leveraged in inference attacks to recover the records and queries. We also discussed the possible countermeasures to withstand these attacks. The details are given in Chapter 2. In particular, in Chapter 3, we present three different methods to hide the size pattern and break the link between the size and search patterns.

Second, we have designed and implemented three different specific SE schemes that achieve the above goals:

- **ObliviousDB**. We design a secure PEKS scheme built on top of the hybrid cloud infrastructure, called *ObliviousDB*. *ObliviousDB* is applicable for organisations with a hybrid cloud infrastructure, where a trusted private cloud server is deployed between users and the public CSP. According to the latest report given by RightScale [44], 51% of the enterprises are using the hybrid strategy. Basically, to protect the search pattern and data distribution in databases, *ObliviousDB* randomises both the encrypted queries and records. The bilinear pairing technique enables the CSP to perform the search operation between randomised queries and records. By using the proxy re-encryption primitives [4, 45], *ObliviousDB* supports MwMr access. Furthermore, compromised users can be revoked efficiently without updating keys and re-encrypting records. In addition, by delegating part of the storage and computation overhead to the private cloud, both the forward and backward privacy is guaranteed. Meanwhile, the leakage of access and size patterns can be controlled with a tradeoff on performance. We implemented a prototype of the scheme and tested its performance. The experiment result shows its practical efficiency. This part of the thesis has been published in [46]. We present this scheme in Chapter 4.

- **Multi-cloud encrypted database (*P-McDb*)**. *P-McDb* is our second solution achieving

the above goals. Unlike *ObliviousDB*, *P-McDb* utilises a multi-cloud strategy that involves multiple untrusted public CSPs and does not rely on any trusted entity. Rightscale's latest report shows that 81% of the investigated enterprises use multi-cloud strategy, and 20% of them use multiple untrusted public CSPs [44]. Thus, *P-McDb* is feasible to most of the enterprises. Roughly speaking, *P-McDb* employs two non-colluding CSPs. One is responsible for storing the encrypted database and performing encrypted search operations, and the other one is in charge of managing the auxiliary information used for searching and ensuring higher security guarantees. As long as the two CSPs do not collude together, the confidentiality of the records and queries are ensured even if the CSPs mount inference attacks. Moreover, *P-McDb* also supports multi-user access, and it can revoke any user efficiently without updating the keys and re-encrypting the records. Another important aspect is, *P-McDb* is SSE scheme, and it is built based on only symmetric encryption and XOR operations, making it much more efficient than HE-based and ORAM-based SE solutions, such as [41, 42, 47, 48]. This part of the thesis has been published in [49] and described in Chapter 5.

- **SGX-assisted encrypted database**. The last solution presented in this thesis combines SSE with Intel Software Guard Extension (SGX) [50], a trusted hardware that can be embedded in the CSP. Ideally, if SGX can store the whole encrypted database and process all the queries, the CSP will learn nothing. Unfortunately, SGX suffers from very limited memory space and side channel attacks. Our solution addresses the two limitations. Basically, B+ tree structure is leveraged in this solution to support sub-linear search. Moreover, SGX loads and processes the tree indices in batches. On the one hand, this method ensures that the access pattern is protected. On the other hand, SGX can also process a large database without exhausting its memory. To mitigate side channel attacks, the B+ tree is searched in a balanced way, independent of the query and the access pattern. Furthermore, apart from the simple equality and range queries, this solution also allows the CSP to perform aggregates, such as SUM, GROUP BY and ORDER BY, join and other complex queries. To analyse the performance, we evaluate our proposed scheme on Big Data benchmark [51] and compare it with ObliDB [52], an SGX-assisted scheme guaranteeing the privacy of access pattern. Our scheme outperforms ObliDB by at least 5.78×. We also compare our scheme with a baseline implementation without access pattern protection but with sub-linear search support. Our results show that our techniques to resist side channel attacks add less than 27× overhead when the B+ tree contains 1 million keys. This scheme has been published in [53] and presented in Chapter 6.

1.5 Organisation

The rest of this thesis is organised as follows. In Chapter 2, we formally define the leakage in existing SE schemes for databases and summarise the leakage-based attacks discovered in recent years. Moreover, we discuss the possible ways to minimise the leakage and defend against the attacks. In Chapter 3, we present several different approaches to avoid the size pattern leakage. In Chapter 4, 5, and 6, we present the design and implementation details for the proposed SE schemes: *ObliviousDB*, *P-McDb* and the SGX-assisted encrypted database, respectively. Finally, in Chapter 7, we conclude the thesis and discuss the future research directions.

Chapter 2

Leakage and Attacks

In this thesis, we mainly aim at minimising information leakage in SE schemes and resisting leakage-based inference attacks. Before giving any solution, in this chapter, we first summarise and define the leakage in existing SE schemes for databases. Second, we revisit the techniques used in existing inference attacks. Finally, based on the leakage and techniques used in inference attacks, we give possible countermeasures to defend against such attacks.

2.1 Notations

In this section, we define the notations used throughout this thesis.

A database $DB = \{rcd_1, ..., rcd_N\} = \{(e_{1,1}, ..., e_{1,F}), \ldots, (e_{N,1}, ..., e_{N,F})\}$ is a $N \times F$ two-dimension array with N rows corresponding to records and F columns corresponding to fields or attributes. Each record $rcd_{id} = (e_{id,1}, ..., e_{id,F})$ consists of F elements and is identified with an unique identifier id.

We define $D_f = \bigcup_{id=1}^{N} e_{id,f}$ as the keywords universe of field f, and define $d_f = |D_f|$ as its cardinality, where $1 \leq f \leq F$. Let $\mathbf{D} = \bigcup_{f=1}^{F} D_f$ be the set of all different elements in DB. Assume $d = |\mathbf{D}|$ and $\mathbf{D} = \{e_1, ..., e_d\}$. For each element $e \in \mathbf{D}$, $DB(e) = \{id | e \in rcd_{id}\}$ is the set of identifiers of the records that contain e. $O(e) = |DB(e)|$ is the occurrence of e in DB, *i.e.,* the number of records containing e. The data distribution DB can be presented as $\Gamma = \{DB(e_1), ..., DB(e_d)\}$.

Let EDB be the encrypted version of DB, and $|EDB|$ be the number of encrypted records in EDB. Each encrypted record is represented as $Ercd$. The id of an encrypted record indicates its storage location in EDB. Each element e is encrypted into e^*. $\Gamma^* = \{EDB(e^*), ...\}$ is the data distribution of EDB, where $EDB(e^*) = \{id | e^* \in Ercd_{id}\}$. In particular, if e^* is semantically secure, $|EDB(e^*)| = 1, \forall e^* \in EDB$.

We say EDB is *static* if it only supports select queries, *i.e.,* no update, insert and delete

queries are ever expected. In static databases, the distribution Γ^* does not change. On the contrary, if EDB allows users to insert new records, delete or update old records after bootstrapping, we say it is *dynamic*. In a dynamic database, Γ^* varies with delete, insert and update queries.

We use Q and EQ to represent a query and an encrypted query, respectively. In this thesis, we focus on simple queries with only one predicate each. Thus, we formalise a SQL query as $Q = (type, f, operator, e)$. $Q.type$ represents the operation of the query, which can be 'select' and 'delete' for dynamic databases but can only be 'select' for static databases. The 'insert' queries are represented by encrypted records, and the 'update' queries can be performed by deleting the old records first and inserting the updated records afterwards. $Q.f$ is the field identifier to be searched, and $Q.e$ is the interested keyword. $Q.operator$ is '=' for equality queries, and $Q.operator \in \{=, <, >, \leq, \geq\}$ for range queries. An encrypted query is defined as $EQ = \{type, f, operator, e^*\}$. In particular, for any two queries Q_i and Q_j, we say they are in the *same structure*, if $Q_i.type = Q_j.type$, $Q_i.f = Q_j.f$ and $Q_i.operator = Q_j.operator$, and $Q_i = Q_j$ if $Q_i.e = Q_j.e$ as well. On the contrary, $Q_i \neq Q_j$ if one of the attributes is unequal.

We say $EQ(Ercd) = 1$ when $Ercd$ matches EQ. $EQ(EDB) = \{(id, Ercd_{id}) | EQ(Ercd_{id}) = 1\}$ stands for the search result of EQ, which can be alternatively represented as $SR(EQ)$.

As mentioned in Chapter 1, most of the SE schemes suffer from the leakage of the search, access, and size patterns. Moreover, dynamic SE schemes may also fail to ensure the forward and backward privacy. Here we give formal definitions for those properties.

Definition 1 (**Search Pattern**). Given a sequence of q queries $(Q_1, ..., Q_q)$, the search pattern represents the relationship between any two queries Q_i and Q_j, *i.e.,* $Q_i = Q_j$ or $Q_i \neq Q_j$, where q is a polynomial size and $1 \leq i < j \leq q$.

Definition 2 (**Access pattern**). Given a sequence of q queries $(Q_1, ..., Q_q)$, the access pattern represents the records matching each query, *i.e.,* $(SR(Q_1), ..., SR(Q_q))$.

Definition 3 (**Size pattern**). Given a sequence of q queries $(Q_1, ..., Q_q)$, the size pattern represents the number of records matching each query, *i.e.,* $(|Q_1(DB)|, ..., |Q_q(DB)|)$, or $(|SR(Q_1)|$, ..., $|SR(Q_q)|)$.

Definition 4 (**Forward Privacy**). Let $Ercd_t$ be an encrypted record inserted or updated at time point t, a dynamic SE scheme achieves forward privacy, if $EQ(Ercd) = 0$ is always true for any query EQ issued before t.

Definition 5 (**Backward Privacy**). Let $Ercd_t$ be an encrypted record deleted at time point t, a dynamic SE scheme achieves backward privacy, if $EQ(Ercd) = 0$ is always true for any query EQ issued after t.

In Table 2.1, we summarise a number of SE schemes proposed in the literature based on whether they achieve the properties or not. These schemes either aims at supporting rich func-

Table 2.1: A comparison of SE schemes.

Schemes	**Search pattern**	**Access pattern**	**Size pattern**	**Forward privacy**	**Backward privacy**	**Key management**	**Remark**
Curtmola *et al.* [8] SSE-1	×	×	×	Static	Static	○	SSE
Curtmola *et al.* [8] SSE-2	×	×	×	Static	Static	◐	SSE
Cash *et al.* [31]	×	×	×	Static	Static	○	SSE
Jarecki *et al.* [54]	×	×	×	Static	Static	○	SSE
Bösch *et al.* [55]	✓	×	×	Static	Static	○	SSE
Cao *et al.* [5]	✓	×	×	Static	Static	◐	SSE
Wang *et al.* [43]	✓	✓	✓	Static	Static	◐	HE-based
Ishai *et al.* [41]	✓	✓	✓	Static	Static	○	ORAM-based
Chen *et al.* [56]	–	✓	✓	Static	Static	○	SSE
Hang *et al.* [12]	×	×	×	×	×	●	PEKS
Ferretti *et al.* [9]	×	×	×	×	×	●	SSE
Popa *et al.* [17]	×	×	×	×	×	●	SSE
Sarfraz *et al.* [18]	×	×	×	×	×	●	SSE
Sun *et al.* [24]	×	×	×	×	×	●	PEKS
Yang *et al.* [22]	×	×	×	×	×	●	PEKS
Asghar *et al.* [4]	×	×	×	×	×	●	PEKS
Bao *et al.* [57]	×	×	×	×	×	●	PEKS
Popa *et al.* [58]	×	×	×	×	×	●	PEKS
Tang [59]	×	×	×	×	×	●	PEKS
Kiayias *et al.* [60]	×	×	×	×	×	●	PEKS
Cash *et al.* [6]	×	×	×	×	×	●	SSE
Kamara *et al.* [14]	×	×	×	×	×	○	SSE
Kamara *et al.* [15]	×	×	×	×	×	○	SSE
Hahn *et al.* [11]	×	×	×	×	×	○	SSE
PPQED [42]	×	×	×	×	×	○	HE-based
Naveed *et al.* [16]	×	×	✓	×	×	○	SSE
Stefanov *et al.* [19]	×	×	×	✓	×	○	SSE
Chang *et al.* [7]	×	×	×	✓	×	○	SSE
Rizomiliotis *et al.* [61]	×	×	×	✓	×	○	ORAM-based
Bost [32]	×	×	×	✓	×	○	SSE
Song *et al.* [62]	×	×	×	✓	×	○	SSE
Bost *et al.* [34]	×	×	×	✓	✓	○	SSE
Zuo *et al.* [35]	×	×	×	✓	✓	○	SSE
Sun *et al.* [63]	×	×	×	×	✓	○	SSE
Hoang *et al.* [64]	✓	✓	×	✓	✓	○	SSE
$PPQED_a$ [42]	✓	✓	×	✓	✓	○	HE-based
Our objectives	✓	✓	✓	✓	✓	●	–

✓ and × indicate that the property is protected or not, respectively. ○ represents a Single User (SU) scheme. ◐ represents SwMr scheme. ● represents a MwMr scheme.
Static means the scheme does not support insert, update, or delete operations.

tionalities, like multi-keyword search, dynamic operation or multi-user access, or supporting sub-linear search efficiency. Unfortunately, most of them suffer from leakage even if they have been proved secure formally or informally under different security models. In this thesis, we aim at proposing SE schemes that can resist leakage-based attacks by protecting the size, search and access patterns, and ensuring both the forward and backward privacy. Meanwhile, the proposed SE schemes should be dynamic and have flexible key management mechanisms to support multi-user access. The proposed schemes could be SSE or PEKS. In the rest of this thesis, we will give more details.

2.2 Leakage Level Definition

In [27], Cash *et al.* define four different levels of leakage profiles for encrypted file collections according to the method of encrypting files and the data structure supporting encrypted search. Yet, we cannot apply these definitions into databases directly, since the structure of a file is different from that of a record in the database. In particular, a file is a collection of related words arranged in a semantic order and tagged with a set of keywords for searching, whereas, a record consists of a set of keywords with pre-defined attributes. Moreover, a keyword may occur more than once in a file and different keywords may have different occurrences, whereas, a keyword of an attribute generally occurs only once in a record. Starting on the leakage layers defined in [27], in this section, we provide our own layer-based leakage definition for encrypted databases. Specifically, we use the terminology leakage profile to describe the information the CSP can learn about the data directly from the encrypted database and also the information about the results and queries that can be learned during the access to the database.

The simplest type of SE scheme for databases is encrypting both the records and queries with Property-Preserving Encryption (PPE), such as the DETerministic (DET) and Order-Preserving Encryption (OPE). In DET-based schemes, the same data has the same ciphertext once encrypted, *i.e.,* $e_i^* = e_j^*$ if $e_i = e_j$ and in OPE-based schemes, the encrypted data has the same order as their plaintext, *e.g.,* $e_i^* > e_j^*$ if $e_i > e_j$. In this type of SE scheme, the CSP can check whether each record matches the query efficiently by just comparing their ciphertext. However, these solutions have the greatest leakage. Specifically, in DET-based schemes, such as CryptDB [17] (when the records are protected only with the PPE layer), DBMask [18] and Cipherbase [65], before performing any query, the CSP can learn the data distribution, *i.e.,* the number of distinct elements and the occurrence of each element, directly from the ciphertext of the database. Formally, we say the data distribution of DB is leaked if e^* and e have the *same occurrence*, *i.e.,* $O(e) = O(e^*)$, for each $e \in DB$, where e^* is the ciphertext of e in EDB. To support encrypted range queries, [17, 18, 65] also use OPE primitives, which reveal the order of numerical data elements. In this thesis, we focus on equality queries. Therefore, we define

this leakage profile set as:

- $\mathcal{L}3 = \{\Gamma\}$.

where Γ is the data distribution of the database and defined as $\Gamma = \{DB(e_1), ...DB(e_d)\}$.

The second type of SE for databases encrypts the data with semantically secure primitives, but still encrypts the queries with DET encryption. By using semantically secure primitives, the encrypted data is randomised. In this kind of scheme, the CSP can also perform search operations efficiently by repeating the randomisation over the DET query and then comparing it with the randomised data, as done in [11], Arx [66] and BlindSeer [67]. By encrypting the data element with semantically secure encryption, $O(e^*) = 1$ for each $e^* \in EDB$. Thus, both the data distribution and the order of numbers are protected. However, after executing a query, the CSP could still learn which and how many records match the query, *i.e.,* the access and the size patterns. Moreover, due to the DET encryption, whether the same query is repeated or not, *i.e.,* the search pattern, is also leaked. We define its leakage profile as:

- $\mathcal{L}2 = \{$*size pattern, search pattern, access pattern*$\}$,

Note that without performing any query, PPE-based databases has $\mathcal{L}1$ leakage. After performing queries, PPE-based databases also leak the profiles included in $\mathcal{L}2$.

A more secure SE solution leverages ORAM [37, 38] or combines HE with oblivious data retrieval to hide the search and access patterns. For instance, the HE-based $PPQED_a$ proposed by Samanthula *et al.* [42] and the ORAM-based SisoSPIR given by Ishai *et al.* [41] hide both the search and the access patterns. Specifically, to hide the data distribution and the search pattern, both $PPQED_a$ and SisoSPIR randomise the encrypted records and queries. To conceal the access pattern from the CSP, in $PPQED_a$, the match result between records and queries is encrypted and can only be learned by the user. Furthermore, the matched records are loaded through oblivious transfer protocols. Thus, the CSP never learns which records match the query. In SisoSPIR, whether each record matches the query is checked by the user, and the access pattern is protected by using ORAM. Unfortunately, in both schemes, the CSP can still learn how many records are returned to the user for each query, *i.e., the communication volume*. According to [29], the HE-based and ORAM-based SE schemes have fixed communication overhead between the CSP and users. Specifically, the length of the message sent from the CSP to the user as the result of a query is proportional to the number of records matching the query. Based on this observation, the CSP can still infer the size pattern. Thus, the HE-based and ORAM-based SE schemes are vulnerable to size pattern-based attacks, such as the count attack. The profile leaked in HE-based and ORAM-based SE schemes can be summarised below:

- $\mathcal{L}1 = \{$*communication volume*$\}$.

In addition to the leakage profiles defined above, the majority of existing SE schemes also leak some basic information. Indeed, in order to maintain the search functionality of encrypted records, most SE schemes, such as [17] and [4], only encrypt the interested table name, field name and keyword involved in the query, whereas, the query type and query operator are kept in plaintext. Furthermore, even if the interested field in the query is encrypted, the CSP can still learn if any two queries search over the same field or not, called *field pattern*. since the schema of the database is fixed and the CSPs can learn which field is searched for each query. Formally, we define the *field pattern* as:

Definition 6 (**Field Pattern**). Given a sequence of q queries $(Q_1, ..., Q_q)$, the field pattern represents the relationship between any two queries fields, *i.e.,* whether $Q_i.f = Q_j.f$ or $Q_i.f \neq Q_j.f$.

Herein, we define this set of basic leakage as:

- $\mathcal{L}0 = \{$*query type, query operator, field pattern*$\}$.

Table 2.2: Summary of leakage profiles and attacks against encrypted databases.

Leakage	Schemes	Attacks
$\mathcal{L}3$	CryptDB [17] DBMask [18] Cipherbase [65] Monomi [68] Seabed [69]	Frequency analysis attack IKK attack Count attack Record-injection attack
$\mathcal{L}2$	Asghar *et al.* [4] Blind Seer [70] [67] Arx [66] PPQED [42]	IKK attack Count attack Record-injection attack
$\mathcal{L}1$	$PPQED_a$ [42] SisoSPIR [41]	Count attack
$\mathcal{L}0$	*ObliviousDB* [46] *P-McDb* [49] SGX-assisted encrypted database [53]	N/A

2.3 Attacks against SE Solutions

With the leakage levels defined in Section 2.2, the SE schemes are still provable secure under either the real-ideal [7, 8], Universal Composable (UC) [71] or other specific security models. However, in practice, the leakage can be exploited to recover the content of records and queries. In recent years, a long line of research on leakage-based attacks has been proposed in the literature. Table 2.2 summarises the existing SE solutions for relational databases and the attacks applicable to them. In this section, we illustrate how the existing leakage-based attacks could recover the data and queries, including the *frequency analysis attack* introduced in [26], the *IKK attack* proposed in [41], the *file-injection attack* presented in [28], the *record-injection attack* defined in [72], the *count attack* introduced in [27], and the *reconstruction attack* proposed in [29]. Specifically, for each attack, we analyse its leveraged leakage, required knowledge, process, and consequences.

2.3.1 Frequency Analysis Attack

In [26], Naveed *et al.* describe an attack on PPE-based SE schemes, where the CSP could recover the items in encrypted databases by analysing the leaked frequency information, *i.e.,* the data distribution. To succeed in this attack, in addition to the encrypted database, the CSP also requires some auxiliary information, such as the application details, publicly available statistics and prior versions of the targeted database. In practice, such auxiliary information could be obtained by accessing the application, from documentations or through a prior data breach. Moreover, publicly available statistics, *e.g.,* census data or hospital statistics, can be obtained online. In PPE-based SE schemes, the frequency information of an encrypted database is same

as that of the database in plaintext. By comparing the leaked frequency information with the obtained statistics relevant to the application, the CSP could recover the encrypted items stored in encrypted relational databases. For instance, most of the organisations, such as universities, publish their staff lists in their homepages, from which the CSP can learn the statistics of names in the targeted organisations. Moreover, the CSP knows the owner of each encrypted database by nature. By comparing the leaked frequency information with the owner organisation's statistics of names, the CSP can infer the name of each record in the encrypted database. Specifically, the field whose element distribution is close to the known name distribution could be 'Name'. Furthermore, the element with the highest occurrence could be the most popular name in the organisation, and so forth. In [26], Naveed *et al.* recovered more than 60% of records when evaluating this attack with real electronic medical records using CryptDB. We stress that this attack does not require any queries or interaction with users. The encrypted databases with $\mathcal{L}3$ leakage profile, *i.e.,* PPE-based databases, such as CryptDB and DBMask, are vulnerable to this attack.

2.3.2 IKK Attack

IKK attack proposed by Islam *et al.* [25] is the first attack exploiting the access pattern leakage in the literature. The goal of the IKK attack is to recover encrypted queries in encrypted file collection systems, *i.e.,* recover the plaintext of searched keywords. As required in the frequency analysis attack, the CSP also needs to know some background knowledge of the targeted dataset, and then the CSP could guess the possible keywords in the dataset and the expected probability of any two keywords appearing in a file (*i.e.,* co-occurrence probability). Again, obtaining such information is not a hard task if the CSP knows the application scenario. For instance, if the targeted files are from a university, 'lecture', 'workshop' and 'tutorial' could be the possible keywords. The CSP can also simulate the expected co-occurrence probability of any two keywords by carrying out a probabilistic analysis over publicly available online datasets, such as the news and documents published in the univeristy's website. Formally, the CSP guesses m potential keywords and builds an $m \times m$ matrix $\tilde{C}$ whose element is the co-occurrence probability of each keyword pair. The CSP mounts the IKK attack by observing the access pattern revealed by the encrypted queries. Specifically, by checking if any two queries match the same files or not, the number of files containing any two searched keywords (*i.e.,* the co-occurrence rate) can be reconstructed. Assume the CSP observes n queries. It can constructs an $n \times n$ matrix C with their co-occurrence rates. By using the simulated annealing technique [73], the CSP can find the best match between $\tilde{C}$ and C and maps the encrypted keywords to the guesses. In [25], Islam *et al.* mounted the IKK attack over the Enron email dataset [74] and recovered 80% of the queries with certain vocabulary sizes.

The encrypted relational databases with leakage profile $\mathcal{L}2$ or $\mathcal{L}1$, such as Arx [66], Blind

seer [70], and PPQED [42], are also vulnerable to the IKK attack. To mount the IKK attack on databases, the CSP needs to guess the co-occurrences of any two elements in two different fields in the real database based on some background knowledge of the application. The encrypted queries can be recovered by observing the revealed access pattern and comparing them with the known dataset.

2.3.3 File-injection and Record-injection Attack

The file-injection attack [28] is the an active attack mounted on encrypted file collections, which is also named as *chosen-document attack* in [27]. The file-injection attack attempts to recover encrypted queries by exploiting the access pattern in encrypted file storage. Compared with the IKK and the count attack (will be discussed in Section 2.3.4), much less auxiliary knowledge is required: the CSP only needs to know the keywords universe of the system. In [28], Zhang *et al.* presented the first concrete file-injection attack on file storage systems and showed that the encrypted queries can be revealed with a small set of injected files. The details of this attack is given below. The CSP sends files composed of the keywords of its choice, such as emails, to users who then encrypt and upload them to the CSP, which are called *injected files*. If no other files are uploaded simultaneously, the CSP can easily know the storage location of each injected file. Moreover, the CSP can check which injected files match the subsequent queries. By injecting enough files with different keyword combinations, the CSP could recover the keywords included in queries by checking the keywords included in the matched and unmatched injected files. Specifically, the keywords included in the matched but not included in the unmatched injected files are the possible searched keywords. For instance, if the injected files matching a query Q all contain w_1 and w_2, but w_2 is also contained in other injected files that do not match Q, the keyword involved in Q must be w_1.

More recently, Abdelraheem *et al.* [30] mounted a similar active attack on encrypted relational databases with $\mathcal{L}2$ or $\mathcal{L}3$ leakage profiles, called *record-injection attack*. In file storage systems, it is hard to recover the content of the whole file by mounting file-injection attacks, since only the keyword is encrypted with SE. However, in databases, many elements of a record might be encrypted with SE in order to support the search operation over these fields. Abdelraheem *et al.* showed that the record-injection attack could not only recover encrypted queries but also fully recover encrypted records when the elements in all fields are encrypted with SE.

2.3.4 Count and Relational-count Attack

The count attack is proposed by Cash *et al.* in [27] to recover encrypted queries in file storage systems based on the access pattern and size pattern leakage. As in the IKK attack scenario, the CSP is also assumed to know an $m \times m$ matrix $\tilde{C}$ where its entry $\tilde{C}[w_i, w_j]$ holds the co-

occurrence rate of keyword w_i and w_j in the targeted dataset. In order to improve the attack efficiency and accuracy, the CSP is assumed to know, for each keyword w, the number of matching files $count(w)$ in the targeted dataset. The CSP mounts the count attack by counting the number of files matching each encrypted query. For an encrypted query EQ, if the number of its matching files (*a.k.a.* the result size $|SR(EQ)|$) is unique and equals to a known $count(w)$, the searched keyword included in EQ must be w. However, if the result size of a query EQ' is not unique, all the keywords with $count(w) = |SR(EQ')|$ could be the candidates, and it is hard to determine which one of them is correct only based on the result size. Recall that the CSP can construct another matrix C that represents the observed co-occurrence rate between any two queries based on the leakage of access pattern. By comparing C with $\tilde{C}$, the candidates for the queries with non-unique result sizes can be reduced. Specifically, the CSP can discard the wrong keyword candidates for EQ' by comparing the observed $C[EQ', EQ_i]$ with $\tilde{C}$, where EQ_i represents a previously recovered query. Formally, the candidates set for EQ' can be reduced to $\{w' : \tilde{C}[w', w_i] = C[EQ', EQ_i]\}$, where w_i is the keyword of EQ_i. With enough recovered queries, it is possible to determine the keyword of EQ'. In [27], Cash *et al.* tested the count attack against Enron email dataset and successfully recovered almost all the queries.

In [72], Abdelraheem *et al.* applied the count attack on relational databases and named it the *relational-count attack*. The SE solutions for relational databases with leakage profiles above $\mathcal{L}1$ are vulnerable to the relational-count attack. The specific leakage leveraged in this attack is the size pattern, *i.e.,* the number of records matching each encrypted query. Due to the fact that the co-occurrence rate of the elements in the same field must be zero, the relation-count attack does not assume that the CSP has any knowledge about the co-occurrence rate between element pairs. Unlike the file collection, in relational databases, the candidates for a query with non-unique result size are limited to the elements in the searched field. Moreover, which field is searched for each query can be easily learned by the CSP. Therefore, with the knowledge of data elements in plaintext and their frequency information in the real database, the CSP could infer the content of the encrypted queries by only observing their size patterns.

2.3.5 Reconstruction Attack

In ORAM-based systems, such as SisoSPIR proposed by Ishai *et al.* [41], the size, and access patterns are concealed. Unfortunately, Kellaris *et al.* [29] observe that the ORAM-based systems have fixed communication overhead between the CSP and users, where the length of the message sent from the CSP to the user as the result of a query is proportional to the number of records matching the query. That is, for a query Q, the size of the communication sent from the CSP to the user is $\alpha|Q(DB)| + \beta$, where α and β are two constants. In theory, by giving two (query, result) pairs, the CSP can derive α and β, and then infer the result sizes of other queries. In [29], Kellaris *et al.* present the *reconstruction attack* that exploits the leakage of

communication volume, and could reconstruct the attribute names in encrypted databases supporting range queries. In this attack, the CSP does not need to have any prior knowledge about the data, or any of the issued queries or their results. The CSP only needs to know the underlying query distribution prior to the attack. Their experiment illustrated that after a certain number of queries, all the attributes can be recovered in few seconds. Since we focus on equality queries in this thesis, we do not give the attack details here. Nonetheless, after recovering the size pattern for each query, the CSP could also mount the count attack on equality queries. Specifically, given the occurrence of each possible keyword, the CSP could recover the queries whose result sizes are unique. The SE schemes with $\mathcal{L}1$ leakage profile are vulnerable to this attack.

2.4 Countermeasures and Challenges

In this section, we investigate the challenges to block the leakage-based attacks mentioned above and give possible countermeasures.

2.4.1 Size Information

In the rest of this thesis, we refer to the data distribution, size pattern, and communication volume as the size information. The size information-based attacks include the frequency analysis attack, the count attack, and the reconstruction attack.

As discussed in Section 2.3, the schemes with $\mathcal{L}3$ leakage are vulnerable to the frequency analysis attack, where the CSP can recover encrypted data by comparing the frequency information with publicly available statistics. To withstand the frequency analysis attack, the data frequency information should be protected from the CSP. A trivial approach is to encrypt the data with semantically secure encryption primitive, as done in [4,66,70]. However, by performing search operations, the CSP can still learn the number of records matching each query. In particular, when the query only involves an equality predicate, the number of returned records stands for the occurrence of the searched keyword in the database. As mentioned in Section 2.3.4, by mounting the count attack, the encrypted queries can be recovered based on the size pattern. That is, semantically secure encryption alone is not effective to prevent the count attack.

To thwart the count attack radically, the number of records matching each query should be protected from the CSP. Private Information Retrieval (PIR) [75,76] is one possible approach to address this issue. PIR allows users to retrieve the data without leaking which data is retrieved. Specifically, in PIR-based SE schemes, such as [77,78], the CSP returns a much larger data set than required to the user, and then the user searches the data set locally to extract the matched data. In this case, the size pattern is protected from the CSP. However, on the one hand,

PIR-based SE schemes increase the computation, storage and communication overhead on the user side. On the other side, for the applications supporting multi-user access with different permissions, PIR-based SE schemes have to ensure users cannot access unauthorised records even if those records are returned.

As mentioned in [27], another possible countermeasure is to mask the data occurrences with *dummy* or *fake* records. Herein, we define the record that consists of a sequence of elements but does not represent a meaningful item or entity as a *dummy record*. If the dummy records are matchable by queries, each search result will contain a number of dummy records. Furthermore, from the CSP point of view, if they are indistinguishable from real data, the number of real records will be protected from the CSP.

However, when using either PIR or dummy records, we should ensure that the communication volume returned to the user is not proportional to real result size, *i.e.,* the number of real records matching the query. Otherwise, as mentioned above, the CSP can still recover the real result size and mount the count attack. To block the count attack, the relationship between communication volume and the real result size should also be upset. In Chapter 3, we give possible solutions to achieve that in details.

2.4.2 Search Pattern

To defend against the leakage-based attacks, it is also necessary to protect the search pattern from the CSP. The search pattern indicates the relationship among the underlying keywords of queries, *i.e.,* if any two queries involve the same keyword or not. Informally, we say the queries are *indistinguishable* or the search pattern is protected if the CSP cannot tell if they involve the same keywords or not. If the search pattern is not protected, the CSP can learn if users are searching for the same thing or not. Furthermore, when the CSP already recovers a set of queries, it could learn what the users are searching directly based on the search pattern.

To protect the search pattern, the query must be encrypted with semantically secure encryption, which makes the same queries look different once encrypted. Otherwise, the search pattern is revealed directly from the ciphertext of encrypted queries. However, encrypting the query with semantically secure encryption alone is not enough to protect the search pattern. The search pattern can still be inferred from the size and access patterns. For instance, when the result sets for two queries are in the same size, the CSPs can infer the two queries are equivalent with a high probability. In particular, if the result size for each query is distinct, the queries with the same result sizes must be the same. To prevent the CSP inferring the search pattern from the size pattern, one possible solution is to ensure all the queries always match the same number of records. An alternative method is to ensure all the queries always match different numbers of records even when the same queries are repeated.

Unfortunately, even if the encrypted queries are semantically secure and all the queries

match with the same number of records, the CSP could still infer the search pattern by looking at the access pattern. That is, by looking at the storage locations of the encrypted data returned by a search, the CSP can infer that two queries are equivalent if the same result sets are returned since generally only the same query gets exactly the same result set. We give the possible solution for this issue in Section 2.4.3.

Recall that to prevent the frequency analysis attack, the encrypted data must be semantically secure. When both the encrypted data and query are semantically secure, it is hard to check if there is a match between the query and the records. To the best of our knowledge, the only solutions in the literature use complex cryptographic primitives, such as the bilinear pairings and HEs. These primitives tend to be much slower than traditional symmetric encryption. So these methods do not scale well when processing the search operation over millions of records. Therefore, the challenge is to find a more efficient way to test equality between semantically secure encrypted data and queries. Addressing this challenge is one goal of this thesis. The details of our solution are available in Chapters 5 and 6.

2.4.3 Access Pattern

As mentioned in Section 2.3, the IKK, count and file/record-injection attacks take advantage of the access pattern leakage. Specifically, in the IKK and count attacks the CSP learns the co-occurrence rate of keyword pairs from the access pattern, and in the file/record-injection attacks the CSP learns if the injected files match the query from the access pattern. Protecting the access pattern from the CSP is one of the hardest problems in SE schemes.

ORAM is a tool proposed to hide the access pattern leakage, and there has been a lot of progress in the efficiency of ORAM schemes. However, it is still difficult to get a practical SE scheme that is based on the ORAM technique. In the following, we give the four main obstacles in using traditional ORAM in SE schemes. First, ORAM requires the user to know the storage address of the data beforehand, which is not the case in SE schemes. On the contrary, in SE schemes, the storage address of the required data is obtained by the CSP via performing encrypted search. It is impractical to store all the addresses on the user side, especially for the applications with thousands of users and resource-constrained devices. In particular, if all the elements only occur once in the database, the storage on the user side will have the same size as the storage on the CSP. Furthermore, it is hard to synchronise the storage over all the users when the database is modified. Second, despite significant recent improvements [79–81], ORAM incurs huge bandwidth, latency, and storage overheads, making it impractical for SE schemes. According to the study by Naveed [82], the naive approach, downloading the whole database and searching locally for each query, is still more efficient than ORAM. Third, ORAM leaks information to users. In an application with fine-grained access control policies, users should only get what they are allowed to access. However, in order to obfuscate the data

access, ORAM returns random data to the user when the required one is already cached. When the returned random data is outside the access of the user, there is a leakage. Last but not least, as illustrated in [29], the ORAM-based SE solutions still leak the communication volume to the CSP.

HE is another technique can be utilised in SE to hide the access pattern. By using HE technique, the match result between encrypted records and queries can be hidden from the CSP, such that the CSP is unable to learn which records match the result. Only the user can decrypt the match result and get the indices of the matched records. Moreover, via ORAM technique or oblivious transfer protocols [16], the matched records can be downloaded in an oblivious manner. Based on the Paillier homomorphic cryptosystem [39], Wang *et al.* [43] present an SE scheme that supports encrypted multi-keyword search and hides the access pattern over the indices of encrypted file collection. In [42], Samanthula *et al.* also introduce an HE-based SE scheme supporting complex query and protecting the access pattern from the CSP. Unfortunately, all the HE-based SE schemes suffer from the impractical efficiency. For instance, the solution presented in [43] takes around 160 seconds when searching over a database with 12000 files. The one given in [42] has worse performance.

By investigating the techniques used in access pattern-based attacks, we have to investigate more practical solutions. As discussed in Section 2.3, the file-injection, IKK, and count attacks leverage the leakage of access pattern. In the record-injection attack, apart from the ability to inject files/records to the database, the CSP is also able to know which injected records match the query and which do not. In both the IKK and the count attacks, in addition to the size pattern leakage, another issue is the CSP can learn the co-occurrence rates of element pairs by checking if any two search results contain the same record or not. Crucially, to resist access pattern-based attacks, on the one hand, we need to make storage locations of the injected records untraceable for the CSP, *i.e.,* not let the CSP learn if the injected records match queries or not. On the other hand, it is necessary to prevent the CSP from learning if any two search results contain the same records or not. Based on these observations, one possible solution is to shuffle and re-randomise searched records by a third entity after executing each query. Specifically, by shuffling and re-randomising the records after each insert query, the CSP cannot trace the injected records, since their ciphertext and storage locations are all changed. Furthermore, CSP is unable to tell if any two results contain the same records or not. Meanwhile, the CSP cannot infer the search pattern from the access pattern since it cannot tell if the search results of any two queries are the same or not. The challenge is that the shuffling and re-randomising operations should not increase the overhead on the user side, meanwhile, the user should be able to get the query result in an acceptable time.

2.4.4 Forward and Backward Privacy

In dynamic databases, the data can be modified when the users issue insert, delete or update queries. Encrypted dynamic databases might leak extra information if they cannot guarantee forward and backward privacy properly. *Forward privacy* means that the CSP cannot learn if newly inserted data or updated data matches previously executed queries. *Backward privacy* means that the CSP cannot learn if deleted data matches new queries. Supporting forward and backward privacy is fundamental to limit the power of the CSP to collect information on how the data evolves over time. For instance, if the encrypted database cannot ensure forward and backward privacy, the CSP could recover all the queries with the file/record-injection attack by executing all the previous queries again over the newly injected records. Similarly, if the CSP learns the plaintext of deleted records, then the queries could also be recovered by checking if they match deleted data.

In the literature, the schemes proposed in [7, 19, 32, 35, 62, 83, 84] dedicate to achieve forward privacy. Unfortunately, all of these proposals require the user to store a set of the latest keys (or data states). The reason is that only the queries encrypted with the latest keys could match records correctly. In multi-user settings, where multiple users could read and write to the database according to the access control policies, if one of the users inserts or updates a new record or file, the keys have to be updated, and then the new keys would have to be distributed to other users. Otherwise, with the previous keys, the other users cannot get the correct result set. The key management issue makes there schemes impractical for multi-user applications. A more flexible approach is needed.

Another issue is related to the fact that these schemes cannot thwart the file/record-injection attack. Indeed, the schemes proposed in [7, 19, 32, 35, 62, 83, 84] are not effective to ensure forward privacy. When a new record *rcd* is inserted, the CSP cannot learn if it matches previous executed queries or not at that moment. However, the CSP can still infer that by executing more queries and checking the access pattern subsequently. Specifically, assume EQ_i and EQ_j are two queries executed before and after inserting *rcd*, respectively. If $EQ_j(EDB) = EQ_i(EDB) \cup rcd$, there is a high probability that EQ_i and EQ_j involve the same keyword and *rcd* matches EQ_i as well. In particular, if *rcd* is injected by the CSP, it can infer the underlying keyword in EQ_i. Therefore, only ensuring the forward privacy is not sufficient to thwart the file/record-injection attack. The access pattern should be protected simultaneously. To ensure the confidentiality of queries and records, backward privacy should also be guaranteed. Unfortunately, to be best of our knowledge, only [35, 85] achieves both forward and backward privacy.

In [34], Bost *et al.* define three levels of backward privacy based on how much metadata leaks about the inserted and deleted records. The highest level, *backward privacy with insertion*

pattern, leaks the records currently matching each query, when they were inserted, and the total number of updates on each matched record. However, ensuring the backward privacy with insertion pattern is still not sufficient to defend against file-injection/record-injection attack. Specifically, the CSP can tell if the matched records were injected by itself based on their insertion times, since the CSP knows when the malicious records were injected, and then recover the queries. Here, we add one more level with less leakage that can resist file-injection/record-injection attacks. We define it as *strong backward privacy*, where the CSP only learns which records match each query, but cannot learn when the records were inserted and deleted, and when and how many times the records have been updated. In short, in this level, the CSP cannot trace the evolution of any records.

Therefore, another challenge to withstand the leakage-based attacks is to ensure both forward and strong backward privacy effectively. In the remaining chapters, we will give the details of our solution.

Chapter 3

Dummy Records Generation

In the previous chapter, we indicate that inserting dummy records into the database is an effective way to defend against the count attack. Moreover, the dummy records can also be used to break the link between the size and search patterns. In this chapter, we provide more details on how dummy records can help in protecting the size and search patterns. Also, we present the details of generating dummy records for both static and dynamic databases.

For clarity, we first present a basic system setting. For specific schemes, the system will be refined and explained in detail. Basically, in our setting, we define three main entities: the administrator (admin), the data user, and the CSP. The admin manages the database. For instance, she bootstraps the database, sets up security parameters, generates secret keys, and revokes compromised users. The user is authorised to read and/or write records by issuing queries to the CSP. In particular, if the user is authorised to insert records into the database, we also call her *data owner*. As defined 1.3, in SwMr and single-user databases, only one user can insert records, meaning there is only one data owner in such systems. On the contrary, MwMr databases have multiple data owners since they allow multiple users to insert records. The CSP is responsible for storing the database and executing queries issued by users. In the following, we will give the responsibilities of admin and data user in detail for generating dummy records.

3.1 Security Requirements

In this section, we list the requirements for protecting the size and search patterns.

Recall that in the count attack, the CSP can recover an encrypted query if the number of its returned records is unique. To prevent the count attack, it is necessary to hide the number of records matching each query from the CSP. In Section 2.4.1, we mentioned that one possible solution to mask the number of matched records is inserting dummy records into the database. However, to protect the size and search patterns effectively, the inserted dummy records should

satisfy several requirements. First of all, the generated dummy records should be able to match queries and indistinguishable from the real ones in the view of CSP. Otherwise, the CSP can still tell the real size pattern for each query. Assume $EQ(DB)$ represents the real records matching the encrypted query EQ and $EQ(EDB)$ represents both the real and dummy records matching EQ when searching over the encrypted database EDB. If a number of dummy records can match EQ, then $|EQ(EDB)| \geq |EQ(DB)|$. Moreover, if the CSP cannot tell how many records in $EQ(EDB)$ are dummy, $|EQ(DB)|$ is protected from the CSP. However, if $|EQ(EDB)| = \alpha|EQ(DB)| + \beta$ for all queries, *i.e.,* there is a fixed linear (or more complicated) relationship between $|EQ(EDB)|$ and $|EQ(DB)|$, the CSP can still infer the size pattern. To thwart the count attack, the total number of returned records for each query, should not be proportional to the number of real records, *i.e.,* there is no fixed relationship between $|EQ(EDB)|$ and $|EQ(DB)|$ for different queries.

Satisfying the above three requirements, the size pattern can be protected from the CSP. However, it is still not sufficient to protect the search pattern. The CSP could still infer if the queries are searching the same data or not by comparing their result sizes. Specifically, the queries are equivalent with a high probability when they match the same number of records. In particular, if the occurrences of all elements in the database are different, the queries matching the same number of records must be the equivalent. The remaining issue is how to break the link between the size and search patterns. To achieve that, all queries should either always match the same number records or match different numbers of records even when the same queries are repeated. Either case can be achieved by managing the dummy records properly.

Besides, to guarantee the lightweight overhead on the user side, the dummy records should be filtered out easily without decrypting them.

Above all, to protect the size and search patterns, the requirements for dummy records can be summarised as below:

- **R1**: The dummy records should be able to match queries.
- **R2**: The encrypted dummy records should be indistinguishable from the encrypted real records.
- **R3**: The number of returned records in total for each query should not be proportional to the number of matched real records.
- **R4**: The queries should always match the same number of records or different numbers of records even when the same query is repeated
- **R5**: The dummy records should be filtered out easily without decrypting them.

To satisfy R1 and R2, we could assemble dummy records by choosing elements from real records, and encrypt them in the same way as real records. Meanwhile, to achieve R5, the

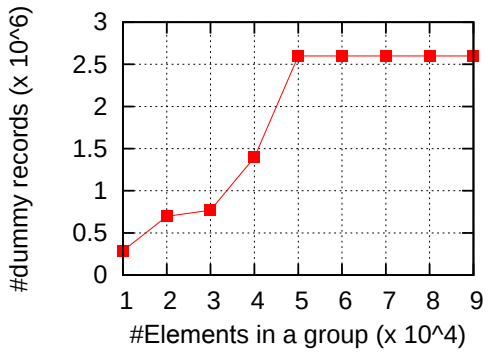


Fig. 3.1. The number of required dummy records with different group sizes for TPC-H 'ORDERS' dataset.

encrypted dummy records should be distinguishable from real ones by users. The details to meet R1, R2, and R5 will be given in the remaining chapters. In this chapter, we focus on R3 and R4. To achieve R3 and R4, the dummy records should be designed based on the real data distribution in the database. Recall that we categorise the databases into static and dynamic based on whether the encrypted database supports insert, delete and update queries. The static database contains a fixed number of stable records, and only supports search queries once it is encrypted and uploaded to the CSP. In other words, the number of elements and their occurrences in static databases are fixed. On the contrary, in dynamic databases, the number of elements and their occurrences can be changed with insert, delete, and update queries. Based on the features of static and dynamic databases, we use different methods to meet R3 and R4.

3.2 Static Databases

For static databases, we assume the admin has all the records and uploads them to the CSP during the bootstrapping. Users can only issue select queries.

As mentioned above, to prevent the CSP inferring the search pattern from the size pattern, all queries should either always match the same number records or always match different numbers of records even when the same queries are repeated. Given that static databases have fixed data distribution, one solution is to ensure that all queries always match the same number of records. This can be done by adding dummy records such that for each field, every element of that field appears in a fixed number of records in the database, *i.e.,* has the same occurrence. In this way, R3 can also be satisfied. That is, the communication volume sent from the CSP to the user is disproportional to the number of matched records. Indeed, the CSP always returns the same volume of data to users no matter what the query is searching for. However, if the occurrences of different elements have huge differences, a large number of dummy records

are required to ensure the elements in each field appear in the same number of records. For instance, in TPC-H benchmark dataset [86], there are 1.5 million records in 'ORDERS' table. For 'CUSTKEY' field, there are 99996 different elements and their occurrences range from 1 to 41. If we ensure all the 'CUSTKEY's appear in 41 records, around 2.6 million dummy records are required, which is almost $2\times$ of the real ones. Consequently, both the storage and search latency increase significantly.

The storage overhead and system performance can be optimised by sacrificing security guarantees. To ease the storage and computation overhead on the CSP, for some scenarios, it is might acceptable to leak some partial information. For instance, in heathcare systems, whether the doctors have the same surnames or not do not conceal much sensitive information about the patients. When generating dummy records, the admin could just pad the doctors with the same surnames into the same occurrences. Consequently, the CSP is unable to distinguish if the queries matching the same number of records involve the same surnames or not, yet it can still tell those matching with different numbers of records must search different surnames. The good aspect is the number of required dummy records can be reduced significantly, such lessens the storage and computation overhead on the CSP.

State-of-the-art. In [87], Bost and Fouque give a theoretical method to minimise dummy records by sacrificing security guarantees. Specifically, they virtually divide the elements into groups and pad those in the same group into the same occurrence with dummy records. As a consequence, the queries involving the elements in the same group will always match the same number of records. The CSP cannot distinguish these queries based on the sizes of their search results and communication volumes. In other words, the CSP is unable to infer the search pattern of the queries in the same group from the size pattern and communication volume. Although the CSP can still tell the queries in different groups are different, the number of required dummy records can be reduced significantly.

Our strategy. For static databases, we use the same strategy as [87] to protect the size and search patterns while minimising the storage and computation overhead on the CSP. However, in [87], Bost only considers the padding for file collections, where the keywords do not belong to different attributes or fields. For relational databases, padding the elements in different fields properly with minimised dummy records is much more complicated. For instance, different fields might need different numbers of dummy records for the padding, making it difficult to assign elements to dummy records as required. Herein, we instantiate this method for relational databases with multiple fields.

Before generating dummy records, the admin first defines the security parameter λ and the groups. Herein, λ stands for the minimum number of elements should be contained in each group. Since the elements in the same group will have the same occurrence, the queries involv-

ing those elements will match the same number of records. Therefore, λ also represents the security level of the search pattern. Formally, for any two queries matching the same number of records, the probability of that they involve the same keyword is $\frac{1}{\lambda}$. Specifically, when $\lambda = 1$, the CSP can tell the queries matching the same number of records must be equivalent. On the contrary, when λ equals to the number of elements in a field, *i.e.,* all the elements in this field are in one group, the CSP is unable to determine if the queries matching the same number of records are equivalent or not. In particular, we also say *the queries are in the same group* if they search for the elements in the same group.

The records can be grouped in many ways. To ease the storage and computation overhead on the CSP, we aim at minimising the required number of dummy records. Indeed, the dummy records are mainly used to pad the occurrence of each element to a threshold. If the occurrence of each element is close to the threshold, the number of required dummy records will be small. Thus, one possible method to minimise the required dummy records is putting the elements with similar occurrences in the same group and takes the highest occurrence as the group threshold. Specifically, the admin first populates the elements in each field (*i.e.,* populates D_f, $\forall f \in [1,F]$), and counts their occurrences. Then, the admin sorts the elements in D_f based on their occurrences in non-decreasing order. From the element with the minimum occurrence to the one with the maximum occurrence, every λ elements are logically in the same group, and the maximum occurrence in each group is its threshold. Formally, if the ordered $D_f = (e_1, ..., e_\lambda, ..., e_{d_f})$, where $O(e_1) \leq ... \leq O(e_\lambda) \leq ... \leq O(e_{d_f})$, there will be $\lfloor \frac{d_f}{\lambda} \rfloor$ groups. Moreover, the i-th λ elements

$$E_{i,f} = \{e_{i\lambda+1}, ...e_{i\lambda+\lambda}\}$$

are in the i-th group $g_{i,f}$ and the threshold of this group is $\tau_{i,f} = O(e_{i\lambda+\lambda})$, where $0 \leq i < \lfloor \frac{d_f}{\lambda} \rfloor$. For each $e_{i\lambda+j} \in E_{i,f}$, $\tau_{i,f} - O(e_{i\lambda+j})$ dummy records are required for its padding, where $1 \leq j \leq \lambda$. Formally, for field f, the total number of required dummy records is

$$\Sigma_f = \sum_{i=1}^{\lfloor \frac{d_f}{\lambda} \rfloor} \sum_{j=1}^{\lambda} (\tau_{i,f} - O(e_{i\lambda+j}))$$

. Note that the last group might contain more than λ elements, and its threshold is $O(e_{d_f})$. However, when $d_f < \lambda$, the group contains less than λ elements. In this case, the admin pads D_f with $\lambda - d_f$ *dummy elements*, which could be meaningless elements out of the domain, *e.g.,* random strings for 'NAME' and integers greater than 200 for 'AGE'.

Recall that to protect the size and search patterns, the elements in the same group should have the same occurrence. After defining the groups for all fields, the admin generates dummy records to pad the occurrence of each element to its group threshold. In this thesis, we do not

consider the query with conjunctive predicates, so we do not make effort to pad the element pairs also into the same occurrence. Therefore, when assigning elements to dummy records, the elements in different D_f can be assembled randomly. However, we consider the queries with singe predicate searching over different fields. To protect the search pattern over all fields, the elements in all fields should be grouped and padded. For the database with multiple fields, the problem is different fields might require different numbers of dummy records. Assume $\Sigma_{max} = \max\{\Sigma_1, ..., \Sigma_F\}$. To ensure the elements in all fields can be padded properly as required, the dummy records inserted in total much be no less than Σ_{max}. Whereas, $\Delta_f = \Sigma_{max} - \Sigma_f$ dummy records will be redundant for field f when inserting Σ_{max} dummy records. For each $e_{i\lambda+j} \in D_f$, $\tau_{i,f} - O(e_{i\lambda+j})$ dummy records should be set to $e_{i\lambda+j}$ in field f. However, there are still Δ_f dummy records unassigned in field f. The CSP can tell the records are dummy if the admin keeps them unassigned. On the contrary, if the admin assigns them also with the elements in D_f, the data distribution in involved groups will be disrupted. Our strategy is to assign a meaningless string that does not belong to any D_f, such as 'NULL', to redundant dummy records. After encryption, 'NULL' will be indistinguishable from other elements. Thus, the CSP cannot distinguish between real and dummy records. Moreover, 'NULL' does not affect the occurrence of other elements. Note that users and the admin can search the records with 'NULL'.

In short, the static database can be bootstrapped in four steps. For each field f, the admin first divides D_f into groups. Second, the admin calculates Σ_f and Σ_{max}. Third, the admin inserts Σ_{max} *empty dummy records*, each of which consists of F 'NULL'. Finally, for each $e_{i\lambda+j} \in D_f$, the admin randomly picks $\tau_{i,f} - O(e_{i\lambda+j})$ dummy records with 'NULL' in field f, and updates their f-th 'NULL' into $e_{i\lambda+j}$. After padding all the elements, a number of inserted dummy records might still have 'NULL' elements, whereas, they do not affect the defined security level (we will give the reason when presenting the specific schemes).

It is worth to mention that the value of λ also affects the number of required dummy records. So when defining λ, the performance requirement and storage limitation should be considered. For instance, in TPC-H benchmark dataset, there are 1.5 million records in 'ORDERS' table. For 'CUSTKEY' field, there are around 100,000 different elements and their occurrences range from 1 to 41. For this field, the number of required dummy records for different λ elements is shown in Fig. 3.1. Specifically, if we set $\lambda = 10,000$, *i.e.,* each group could contain at least $10,000$ elements, the records can be divided into 8 groups, and around 0.3 million dummy records are needed in total. When λ is set to $20,000$, the records can be divided into 4 groups, and around 0.7 million dummy records are required totally. When all the elements are in the same group, around 2.6 million dummy records are required.

3.3 Dynamic Databases

The above solution designed for static databases is not sufficient to minimise information leakage in dynamic databases. Unlike static databases, dynamic databases still support insert, delete and update queries once encrypted and uploaded to the CSP. Thus, the data distribution in dynamic databases might change over time. The size pattern is safeguarded from the CSP as long as dummy records can match queries and are indistinguishable from real ones. However, it is hard to hinder the CSP inferring the search pattern from the size pattern once the data distribution is changed. Specifically, in dynamic databases, it is hard to either ensure the queries always match the same number of records even for the queries in the same group, or ensure the queries could match different numbers of records even if the same queries are repeated. For instance, if the initial database is bootstrapped as done for static databases, when a user inserts a new record after the bootstrapping, the occurrence of each involved element will increase by one and be different from that of other elements. Then, the CSP can learn the queries whose result sizes are different from others must associate with the most recent inserted elements. Even worse, if the record is injected by the CSP as done in the record-injection attack, the CSP can learn the underlying keywords in those queries directly. On the contrary, if all the elements have different occurrences in the initial database, different queries will match different numbers of records. However, if there is no insert, delete or update query executed between two select queries, the CSP can learn which queries are repeated based on their result sizes.

For single-user and SwMr dynamic databases, if the data owner knows the data distribution and the groups, it can ensure the elements always have different occurrences or ensure the elements in the same groups have the same occurrences by adding or removing dummy records when executing insert, delete and update queries. However, when the dynamic database supports multiple users to read and write the records, the situation is more complicated, since it is more difficult to ensure all the users always have the latest data distribution and group information. To prevent the CSP from learning the search pattern from size pattern in MwMr dynamic databases, we present three possible solutions. The first two solutions ensure the queries in the same group always match the same number of records, and the third solution ensures all the queries match different numbers of records even if the same queries are repeated. Note that the solutions for MwMr can also be applied in SwMr and single-user databases.

3.3.1 Solution 1

The motivation of our first solution is to ensure the queries in the same group always match the same number of records. The basic idea is fixing the number of groups and the elements in each group and changing the group thresholds when inserting, deleting or updating records. Basically, the admin is responsible for setting up the security parameters and bootstrapping the

database. Users can execute insert, select, delete and update queries. Meanwhile, users also need to manage the dummy records when inserting, deleting or updating real records.

Database bootstrapping. The admin first sets up the system by defining the security parameter λ, which defines the minimum number of elements in a group. If the initial database is not empty, the admin divides the elements in each field into groups and pads them with dummy records. To minimise the number of required dummy records, the admin could also just put every λ elements with similar occurrences in the same groups, as done in our solution for static databases. However, the grouping method could be different and defined based on the requirement of specific schemes. We will give different grouping approaches that fit for our schemes in Chapter 4. In this section, we assume the database has been divided into groups and we focus on the dummy records management when inserting and deleting records.

To ensure the elements in the same group always have the same occurrence, users need to know which λ elements are in the same group. Thus, the admin sends each enrolled user the elements in each group. For instance, assume the groups for field f are $\{\{e_1,...,e_\lambda\},...,\{e_{(\lfloor\frac{d_f}{\lambda}\rfloor-1)\lambda+1}, ..., e_{d_f}\}\}$. All the users enrolled in the system stores $\{\{e_1,...,e_\lambda\},...,\{e_{(\lfloor\frac{d_f}{\lambda}\rfloor-1)\lambda+1}, ..., e_{d_f}\}\}$ for each field locally.

If the initial database is empty, the admin could pre-define the groups based on the historical dataset and publicly available statistics, and then update the groups over time.

Insert query. Assume the inserted record is $rcd = (e_1,...,e_F)$, and let $g_{i_f,f}$ be the group of e_f, $\forall f \in [1,F]$. If a user inserts rcd into the database, the occurrence of each element e_f will increase by one. For each $f \in [1,F]$, to make e_f indistinguishable from other elements in $g_{i_f,f}$, our first strategy is to increase the threshold of $g_{i_f,f}$ and the occurrences of its other elements also by one. Specifically, each time when a user inserts a real record $rcd = (e_1,...,e_F)$, it generates $\lambda - 1$ dummy records and inserts them into the database together with the real one in a random order, such that the CSP cannot tell which one of them is real. To increase the occurrence of other elements in $g_{i_f,f}$, for each field f, the $\lambda - 1$ dummy records cover the other $\lambda - 1$ elements in $g_{i_f,f}$. Moreover, the elements in the F groups, except $(e_1,...,e_F)$, are assembled randomly to form the dummy records.

Delete query. If $rcd = (e_1,...,e_F)$ is a record to be deleted, instead of removing it from the database, we set it to a dummy record. In this way, the occurrences of the involved elements will not be changed. However, if the system never removes records, the database will increase rapidly. To avoid this, the admin periodically removes the records consisting of F 'NULL' from the database. Note that in this process the admin must ensure the elements in the same groups always have the same occurrences. Specifically, the admin periodically checks if each element in each group is contained in one dummy record. If yes, for each element, the admin updates one dummy record containing it to 'NULL'. As a consequence, the occurrence of all

the elements in the same group will decrease by one, but still is the same. When the dummy records only consist of 'NULL', the admin removes them from the database.

The above solution is used to delete a single record. Indeed, a delete query might need to remove multiple records. Thus, the user cannot issue delete queries to the CSP directly. Instead, the user first gets the records to be deleted by issuing a select query that has the same predicate as the delete query, and then updates the returned records to dummy.

Update query. Update queries are performed by deleting the records with old elements and inserting new records with new elements.

Discussion. In this solution, the number of elements in each group is fixed, which is always equal to or greater than λ. Therefore, this solution guarantees the security level of the search pattern. However, this solution suffers from three main disadvantages. The first disadvantage is that a large number of dummy records are required. As mentioned above, for each insert query, $\lambda - 1$ dummy records should be generated and inserted with the real one. When λ is large, the size of the database will increase rapidly with insert queries. Second, to generate dummy records, users have to store the elements in each group, which increases the storage burden on the user side. In particular, when all the elements only occur once in the database, the user will have the same storage overhead as the CSP. Thus, this solution is not suitable for the databases with big $|\mathbf{D}|$ (where $\mathbf{D} = \{D_1, ..., D_F\}$ is the set of all different elements in the database). Moreover, when the initial database is empty, the occurrence of elements in each group is 0. If the users insert real records one by one, the rate between inserted real and dummy records each time is fixed to $1 : \lambda - 1$. As a result, the CSP can learn there are $\frac{1}{\lambda}$ real records in the database and in each group. In the following, we give two possible ways to eliminate the three disadvantages.

The first solution is that, instead of inserting real records one by one by users, the admin aggregates records from users, generates dummy records and uploads them to the database in batch periodically. By doing so, obviously, the users do not need to store the elements of each group, and only the admin needs to. Moreover, the number of required dummy records can be reduced. In addition, the rate between real and dummy records in the database and in each group can be upset even when the initial database is empty. Thus, the CSP cannot infer the number of real records in the database and in each group. We give the reasons in the following.

To protect the search pattern, the occurrence of the elements in the same group should always have the same occurrence, which means the elements in the same group should increase or decrease by the same number simultaneously. If the real records to be inserted cannot ensure that, a number of dummy records are required. Therefore, the more elements in the same group involved by the real records in each batch, the fewer dummy records are needed to cover the uninvolved elements. For instance, assume n real records $\{(e_{1,1}, ..., e_{1,F}), ..., (e_{n,1}, ..., e_{n,F})\}$

are to be inserted together in a batch, where $n \geq 1$. In the worst case, $(e_{1,1},...,e_{1,F}) = ... = (e_{n,1},...,e_{n,F})$, *i.e.,* the n records have the same element for each field. In other words, for each field f, the n records only involve one group $g_{i,f}$ and cover one element in $g_{i,f}$. In this case, the occurrence of the involved element increases by n, Thus, n dummy records are required to increase each uninvolved element in $g_{i,f}$ also by n. Since, $(\lambda - 1)$ elements in $g_{i,f}$ are not covered, the admin has to generate $n(\lambda - 1)$ dummy records in total, which is equivalent to the number of required dummy records when inserting the n real records one by one. However, in the best case, for each field f, the n records could only involve one group $g_{i,f}$ and cover n different elements in $g_{i,f}$. That is, $\{e_{1,f},...,e_{n,f}\}$ are different elements in group $g_{i,f}$ for all $f \in [1,F]$. In this case, the occurrence of each involved element increases by 1. Then, only $\lambda - n$ dummy records are needed in total to increase the occurrence of other $\lambda - n$ elements in $g_{i,f}$ by one. In general, the number of required dummy records ranges between the best and worst cases, which must be less than that when inserting the real records one by one. Moreover, the number of dummy records inserted each time changes with the data distribution of the real records to be inserted, making the rate between real and dummy records variable.

The second method is to keep the occurrence of each inserted element e_f unchanged by updating one dummy record with e_f to 'NULL', instead of increasing the occurrences of other elements. Specifically, before inserting *rcd*, for each element $e_f \in rcd$, the user first checks if there is a dummy record containing e_f. If yes for all the F elements in *rcd*, the user inserts the record. Then, for each e_f, the user updates one of the dummy records with e_f to 'NULL'. Otherwise, the user increases the occurrences of elements in involved groups as above.

3.3.2 Solution 2

Our second solution is also designed for MwMr dynamic databases. In particular, this method is used to manage the dummy records when grouping the database based on the element occurrences. Comparing with the first solution, this solution aims at reducing the storage overhead on users and the number of required records by weakening the security guarantee for the search pattern. The main idea is to fix the thresholds for all groups but migrate elements among groups by changing their occurrences.

Database bootstrapping. As done in the first solution, if the initial database is not empty, the admin defines λ and the groups, and then pads the occurrence of elements into their group thresholds. The difference is the admin only sends all the group thresholds to enrolled users. When the initial database is empty, the admin pre-defines the keywords universe and group thresholds for each field based on an available historical dataset or public statistics.

Insert query. When inserting $rcd = (e_1,...,e_F)$, the motivation of the user is to ensure the occurrences of $(e_1,...,e_F)$ will not be unique. The main idea is to migrate $(e_1,...,e_F)$ to new

groups by changing their occurrences and generating a certain number of dummy records same as rcd. Specifically, the user first gets $O(e_f)$ by selecting all the records with e_f, for each $f \in [1,F]$. Recall that the user stores all the group thresholds. Second, for each e_f, the user finds the group $g_{min_f,f}$ whose threshold:

$$\tau_{min_f,f} = min\{\tau_{j_f,f} | \tau_{j_f,f} > O(e_f), 1 \leq j \leq \lfloor \frac{d_f}{\lambda} \rfloor\}$$

($\lfloor \frac{d_f}{\lambda} \rfloor$ is the number of groups in field f and $d_f = |D_f|$). In short, $\tau_{min_f,f}$ is the minimum threshold that greater than $O(e_f)$. Third, the user generates:

$$m = max\{\tau_{min_f,f} - O(e_f) | 1 \leq f \leq F\}$$

dummy records. For each field f, $\tau_{min_f,f} - O(e_f)$ of the dummy records will be assigned with e_f, and the other $m - (\tau_{min_f,f} - \tau_{i_f,f})$ dummy records will be assigned with 'NULL'. Finally, the user inserts the $m+1$ records into the database in random order. As a consequence, e_f is migrated to $g_{min_f,f}$ for each $f \in [1,F]$.

Delete query. When deleting $rcd = (e_1, ..., e_F)$, the user sets it to dummy. In this solution, redundant dummy records are also deleted in two phases. Specifically, the admin first periodically checks the occurrence of each element in each group. Assume the real occurrence and occurrence of e_f are $O^{real}(e_f)$, and $O(e_f)$, respectively. Second, the admin finds the minimum group threshold that greater than $O^{real}(e_f)$, *i.e.,*

$$\tau_{min_f,f} = min\{\tau_{j_f,f} | \tau_{j_f,f} > O^{real}(e_f), 1 \leq j \leq \lfloor \frac{d_f}{\lambda} \rfloor\}$$

If $\tau_{min_f,f} < O(e_f)$, $O(e_f) - \tau_{min_f,f}$ dummy records with e_f will be updated to 'NULL' in field f. Meanwhile, the admin periodically removes records with F 'NULL'.

Update query. Update queries are performed by deleting the records with old elements and inserting new records with new elements.

Discussion. In this solution, the number of required dummy records relies on the difference between group thresholds, which could be less than λ. To further reduce the number of dummy records, as mentioned above, the admin can accumulate records and insert them together in one batch, or ensure the occurrence of inserted elements unchanged by updating certain dummy records to 'NULL'. Another important aspect is that this solution adds much less overhead on the user side. Basically, for each group, the users only store the threshold, which is significantly smaller than the elements set of each group. Whereas, the number of elements in each group changes over time, which could be smaller or greater than λ over time, making the number of indistinguishable queries in each group also changes over time. In particular, when a group

contains only one element, the CSP can tell the precise times it has been requested since its occurrence is distinct from others. Recall that the first solution can ensure each group always has λ elements. Therefore, this solution achieves a lower security level than the first solution.

3.3.3 Solution 3

In the first two solutions, the strategy to break the link between the search and size patterns is ensuring a set of queries always match the same number of records. On the contrary, our third solution aims at ensuring the queries could match different numbers of records even if the same queries are repeated. The main idea is inserting a number of dummy records to hide the data distribution and refreshing the dummy records after executing each query. This solution can also be utilised for single-user, SwMr, and MwMr databases.

System setup. The admin first populates D_f for each field and sends it to each enrolled user. Second, the admin defines a parameter δ to control the rate of dummy records in the database. If the initial database is not empty, the admin generates N' dummy records during the bootstrapping, where $N' \in [1, \delta N]$ and N is the number of real records in the database. The elements for each field f of dummy records are picked from D_f randomly.

Insert query. When inserting records, users also generate a number of random dummy records and insert them together with real ones in random order. Specifically, if there are n real records to be inserted, the user generates m dummy records, where $m \in [1, \delta n]$. This solution does not adopt the strategy of groups or pad the occurrence of each element to a certain value. Thus, for each field f, the f-th elements of dummy records can be picked from D_f randomly. In this case, different queries could match the same or different numbers of records. However, when a query is repeated, it matches the same number of records. The CSP can still guess the queries with the same result size could be equivalent with a high probability. To solve this issue, the dummy records should be refreshed after executing each query. In Section 2.4.3, we mentioned that it is necessary to shuffle and re-randomise searched records after executing each query by a third entity to protect the access pattern. The third entity can also be leveraged to refresh the dummy records. Specifically, for each dummy record, the third entity randomly assigns F new elements to it, making the data distribution of dummy records updated. Consequently, if a query is repeated, the result size could be different. Then, the CSP cannot tell if the queries matching the same number of records are equivalent or not.

Delete query. This solution does not need to consider the occurrence of elements. When executing delete queries, the matched records, including both the real and dummy records, can be deleted directly.

Update query. Update queries are performed by deleting the records with old elements and inserting new records with new elements.

Table 3.1: Comparison of the solutions for managing dummy records.

Solution	Property	User Storage	#Dummy Records	Security
Solution 1	Variable group thresholds	Group elements	Large	λ
Solution 2	Variable group elements	Group thresholds	Medium	Variable
Solution 3	No group	Keywords universe	Large	–

Discussion. In this solution, the dummy records are generated randomly. Thus, it simplifies the dummy records management on both the admin and users. However, this solution also requires a significant number of dummy records to protect the search pattern. Ideally, for each encrypted query EQ searching over field f, its interested keyword should be an element in D_f with the same probability. Assume e_1 and e_2 are two elements in D_f, and the number of real records containing them are $O(e_1)$ and $O(e_2)$, respectively. Formally, if the interested keyword involved in EQ is e_1, there should be $x_1 = |EQ(EDB)| - O(e_1)$ dummy records contain e_1. Similarly, $x_2 = |EQ(EDB)| - O(e_2)$ dummy records contain e_2 if EQ involves e_2. To fully protect the search pattern, the two events should happen with very similar probabilities, which means $|\frac{C_{N'}^{x_1}}{|D_f|^{N'}} - \frac{C_{N'}^{x_2}}{|D_f|^{N'}}|$ has to be negligible. To achieve this, N' must be big enough, meanwhile, x_1 and x_2 must be very close. That is, this solution requires a large number of dummy records, and could only protect the search pattern of the queries with approximate result sizes.

3.3.4 Conclusion

In Table 3.1, we summarise and compare the above three solutions based on their storage overhead on the user side, the number of required dummy records and the security level of the search pattern. The first solution ensures a higher level of security, but requires more dummy records and put the most storage overhead on the user side than the other two solutions. In the second solution, both the required dummy records and the storage overhead on the user side are optimised, yet the security guarantee varies with queries, which could be worse than the first solution. The last solution does not group the data, thus it simplifies the dummy records generation on users and the admin. However, it puts the same storage overhead on the user side as the first solution, which is much bigger than solution 2. The security level of the search pattern in this solution varies with the number of inserted dummy records. That is, the more records inserted, the harder for the CSP to infer the search pattern based on the size pattern and communication volume. Above all, all the three solutions have advantages and disadvantages. Which solution should be used can be determined based on the application requirements. For instance, the applications with resource-constrained user side can select the last solution, and the applications require a higher level of security can use the first solution.

Investigating a better way to break the link between the size and search patterns for dynamic

databases with minimum dummy records is still an open problem. In the rest of this thesis, we integrate the first solution with our proposed schemes, and assume the elements in the same group always have the same occurrence. Moreover, the solution might be adjusted slightly to adapt our proposed SE schemes.

Chapter 4

Hybrid Cloud Based Solution: ObliviousDB

A hybrid cloud deployment is a cloud computing environment that combines on-premises private cloud platforms with public cloud services. By allowing workloads to be moved between private and public clouds as computing needs and costs change, the hybrid cloud apporach gives businesses greater flexibility and more data deployment options. Moreover, the private cloud could be considered as a trusted entity, because it is inherently managed by the organisation, where sensitive data can be stored and processed without any extra layer of security. According to the latest report by Rightscale [44], the hybrid cloud computing approach is getting more popular among large enterprises.

In this chapter, we present *ObliviousDB*, our SE scheme that exploits hybrid cloud environments to minimise information leakage and thwart inference attacks. Specifically, *ObliviousDB* not only lessens the leakage of search, access, and size patterns, but also ensures the forward and backward privacy of records. Moreover, by integrating the key management mechanism given in [4, 45], *ObliviousDB* allows multiple users to read and write the database, while it does not need to update the keys or re-encrypt records in case of user revocation. Basically, in *ObliviousDB*, the public infrastructure is used for storing all the records while the private infrastructure is used mainly for running our *Oblivious Private Service (OPS)*, a service for maintaining metadata information about the records stored in the public infrastructure. The OPS plays a major role in ensuring the confidentiality of the data and manages the data structures for achieving search efficiency. In terms of its functionality, the OPS is similar to the proxy server used in CryptDB [17]. However, unlike CryptDB, we have designed the OPS to be robust against attacks, *i.e.,* a compromised OPS will not reveal sensitive data to adversaries.

In summary, *ObliviousDB* makes the following contributions in this chapter:

1. *ObliviousDB* minimises the information leaked to the public CSP when executing queries

by (i) dynamically re-randomising the encrypted data, (ii) shuffling the locations of records within the database, and (iii) introducing and varying a number of dummy records, necessary for achieving the *search and access pattern privacy*.

2. *ObliviousDB* supports both forward and backward privacy by randomising data and query through the use of fresh nonces. In this way, even if the CSP stores a search query, it cannot be matched with new data. Likewise, new queries cannot be executed over deleted records.

3. *ObliviousDB* is a MwMr SE scheme that supports flexible multi-user access. Moreover, revoking users does not require key regeneration and data re-encryption.

4. To show the feasibility of our approach, we have implemented *ObliviousDB* and measured its performance.

The rest of this chapter is structured as follows. In Section 4.1, we give the system and threat model and describe *ObliviousDB* in high-level. The design details of *ObliviousDB* is explained in Section 4.2. Next, we analyse the security of *ObliviousDB* in Section 4.3. The prototype implementation and performance evaluation are given in Section 4.4. In Section 4.5, we review related work in the literature. Finally, we conclude *ObliviousDB* in Section 4.6.

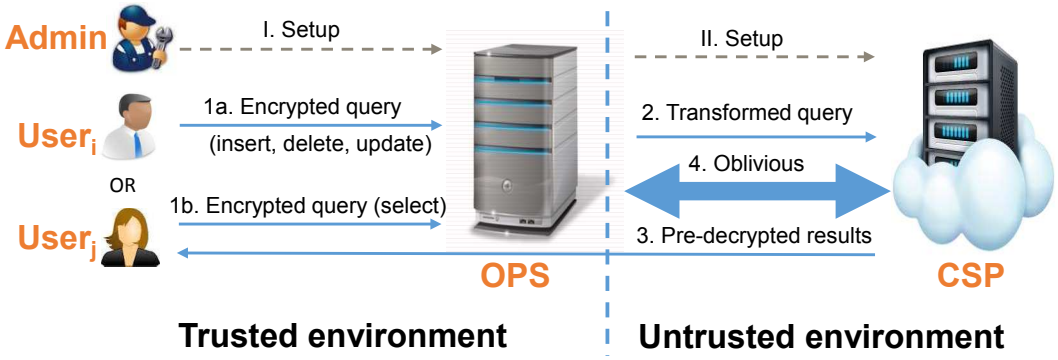


Fig. 4.1. An overview of *ObliviousDB*.
The admin is responsible for running setup (Step I then Step II). A user can insert, delete and update the data (Step 1a) or execute a select query (Step 1b) to receive matching records (Step 3). To control information disclosure, the OPS transforms the query (Step 2) to perform the search followed by an oblivious algorithm (Step 4).

4.1 Overview of *ObliviousDB*

4.1.1 System Model

The system involves four main entities shown in Fig. 4.1:

- **Administrator (admin)**: The admin is responsible for setting up the database, managing users, and regulating access control policies.
- **User**: It represents an authorised user who can execute select, insert, update and delete queries over encrypted data. After executing encrypted queries, a user can retrieve the result set, if any, and decrypt it.
- **Oblivious Private Service (OPS)**: It provides greater security and search efficiency. It serves between users and the CSP. To hide sensitive information about queries, it pre-processes the queries submitted by the user. To improve performance, it manages indexing information. Technically, the OPS is part of the private cloud in the hybrid cloud environment, which is linked with a more powerful public cloud infrastructure.
- **Cloud Service Provider (CSP)**: A CSP is part of the public cloud infrastructure provided by a cloud service provider. It stores the encrypted data and access control policies and enforces those policies to regulate access to the data.

Threat Model. We assume the admin is fully trusted. Users are only considered to keep their keys (and decrypted data) securely. The OPS is deployed in the private cloud, which is owned by the organisation. Hence, we assume the OPS is also trusted. However, it is responsible

for communicating with the external world. Thus, it could be the target of attackers and get compromised, which means the data stored on the OPS could possibly be exposed to attackers.

We consider that the CSP is honest-but-curious. More specifically, the CSP would honestly perform the operations requested by the admin and users according to the designated protocol specification; however, it is curious to analyse the stored and exchanged data so as to learn additional information. We assume that the CSP will not mount active attacks, such as modifying the message flow or denying access to the database.

In this work, we assume that there are mechanisms in place for data integrity and availability. Last but not least, access policy specification is out of the scope of this work, but the approach introduced in [4, 88] can be utilised in *ObliviousDB*.

4.1.2 Proposed Approach

ObliviousDB represents a practical encrypted scheme for an outsourced dynamic database with controllable leakage. Using *ObliviousDB*, queries can be executed without the CSP learning the search, access, and size patterns. The search pattern is protected by using a semantically secure algorithm. The access and size patterns are protected by inserting dummy records and running the oblivious algorithm explained in Section 4.2.5. To support forward and backward privacy, *ObliviousDB* uses fresh nonces to randomise inserted data and queries. To achieve efficiency, *ObliviousDB* divides the data into groups and build an index for each group. In this work, we consider dynamic databases. *Solution 1* given in Chapter 3 is combined with *ObliviousDB* to manage the dummy records when inserting, deleting, or updating records.

In *ObliviousDB*, the admin initialises the system by setting up the OPS (Step I) and the CSP (Step II) as illustrated in Fig. 4.1. After the system is initialised, the admin can add users and generate keying material for them. Using her own key, a user encrypts and sends real and dummy records to the OPS (Step 1a). The OPS first rewrites the records and then sends it to the CSP (Step 2). Another user can perform a select query by encrypting it using her own key (Step 1b). The OPS rewrites the select query to be able to match the records stored in the CSP. Before sending the matched records to the user, the CSP pre-decrypts them in a way without learning the plaintext. However, the returned matched records will contain both real and dummy records (Step 3). Using flags, the user filters out the dummy records and decrypts real records using her own key. Meanwhile, the searched records are sent to the OPS for shuffling and re-randomising (Step 4).

Table 4.1: Data representation in *ObliviousDB*.

(a) *Staff*

Name	**Age**
Alice	25
Anna	30
Bob	27
Bill	25
Bob	33
Baker	25

(b) GDB on the OPS

GID	**Nonce**	**Index List**	**Elements**
$(1, g_{1,1})$	$n_{1,1}$	$\{1,2\}$	$E_{1,1} = \{H_s(Alice), H_s(Anna)\}$
$(1, g_{2,1})$	$n_{2,1}$	$\{3,4,5,6,7,8\}$	$E_{2,1} = \{H_s(Bob), H_s(Bill), H_s(Baker)\}$
$(2, g_{1,2})$	$n_{1,2}$	$\{1,3,4,6,7,8\}$	$E_{1,2} = \{H_s(25), H_s(27)\}$
$(2, g_{2,2})$	$n_{2,2}$	$\{2,5\}$	$E_{2,2} = \{H_s(30), H_s(33)\}$

(c) EDB on the CSP

ID	**1**		**2**	
1	$SE_{n_{1,1}}(Alice)$	$DE(Alice)$	$SE_{n_{1,2}}(25)$	$DE(25)$
2	$SE_{n_{1,1}}(Anna)$	$DE(Anna)$	$SE_{n_{2,2}}(30)$	$DE(30)$
3	$SE_{n_{2,1}}(Bob)$	$DE(Bob)$	$SE_{n_{1,2}}(27)$	$DE(27)$
4	$SE_{n_{2,1}}(Bill)$	$DE(Bill)$	$SE_{n_{1,2}}(25)$	$DE(25)$
5	$SE_{n_{2,1}}(Bob)$	$DE(Bob)$	$SE_{n_{2,2}}(33)$	$DE(33)$
6	$SE_{n_{2,1}}(Baker)$	$DE(Baker)$	$SE_{n_{1,2}}(25)$	$DE(25)$
7	$SE_{n_{2,1}}(Bill)$	$DE(xyz)$	$SE_{n_{1,2}}(27)$	$DE(*)$
8	$SE_{n_{2,1}}(Baker)$	$DE(xyz)$	$SE_{n_{1,2}}(27)$	$DE(*)$

Table (a) is a sample table viewed by users. Table (b) is the group information stored on the OPS. For the first field, we have $GE(Alice) = GE(Anna) = g_{1,1}$ and $GE(Bob) = GE(Bill) = GE(Baker) = g_{2,1}$. For the second field, we have $GE(25) = GE(27) = g_{1,2}$ and $GE(30) = GE(33) = g_{2,2}$. Each group has a nonce to ensure forward and backward privacy, a list of IDs indicating the records in the group, the occurrence threshold, and its elements. The CSP stores Table (c), where each value is encrypted with *SE* and *DE* for data search and retrieval, respectively. Each *SE* value is bound with the nonce of its group. The last two records, consisting of normal *SE* and fake *DE* parts, are dummy.

4.2 Solution Details

4.2.1 Setup

The system is set up by the admin by taking as input a security parameter k. The output is a prime number p, three multiplicative cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of order p, such that there is a "Type 3" bilinear map [89] $\mathfrak{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, which has the properties of *bilinearity*, *computability* and *non-degeneracy*, but there is no symmetric bilinear map defined on $\mathbb{G}_1$ alone or $\mathbb{G}_2$ alone. Let g_1 and g_2 be the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. The admin chooses a random x from Z_p and returns $h = g_1^x$. Next, it chooses a collision-resistant keyed hash function $H : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^l$, and a random key s for H. It also initialises the key store managed by the CSP. That is, $K_S \leftarrow \phi$. Finally, it publishes the public parameters $Params = (\mathfrak{e}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, h, H)$ and keeps securely the master secret key $MSK = (x, s)$.

Building on top of the key management approach given in [4, 45], *ObliviousDB* supports multi-user access with efficient user registration and revocation. Specifically, when the user $user_i$ joining the system, the admin splits MSK into two values x_{i1} and x_{i2}, where $x = x_{i1} + x_{i2}$ mod p and $x_{i1}, x_{i2} \in \mathbb{Z}_p$. Then, the admin transmits $K_{U_i} = (x_{i1}, s)$ and $K_{S_i} = (i, x_{i2})$ securely to $user_i$ and the CSP, respectively. The CSP adds K_{S_i} to its key store: $K_S \leftarrow K_S \cup K_{S_i}$. With K_{U_i}, $user_i$ could issue a query. For revoking a user, we just need to remove K_{S_i} on the CSP.

4.2.2 Group Generation

To protect the search and size patterns, if the initial database is not empty, *ObliviousDB* also divides the data into groups and pads the elements in the same groups into the same occurrence with dummy records. By doing this, the CSP cannot tell if the queries in the same group are searching for the same terms or not. However, after searching, the CSP can still learn if the records and queries are in the same group or not from the size pattern. To avoid this, the admin can pad all the elements in each field into the same occurrence. However, a large number of dummy records are required in this case. In this work, we do not aim to protect the group information from the CSP since the countermeasure is expensive.

Considering the CSP can learn the group information from the size pattern, *ObliviousDB* allows the CSP to only search a group of records for each query, rather than the whole database. By doing so, the query can be processed more efficiently without leaking additional information to the CSP. Yet, in this case, the CSP needs to know which group of records should be searched for each query before searching. To this end, on the one hand, each group should have a unique identifier, and the group identifier of each query should be known. On the other hand, *ObliviousDB* should build indices for the groups.

As mentioned in Section 3.3.1, if all the users store the elements in each group locally,

they can easily know the group identifier for each query. However, storing the elements in each group could put heavy storage overhead on the users. In particular, if each element only occurs once, all the user will have the same storage size as the CSP. Moreover, if the groups are updated by the admin or one user, it is hard to synchronise the storage on all the users in MwMr applications. Fortunately, the OPS is trusted in *ObliviousDB*. Therefore, the admin could store the elements in each group in the OPS and allows the users to access when required. Moreover, in case the OPS is compromised, the elements should be hashed or encrypted.

As an alternative, instead of grouping the elements based on their occurrences, the admin could generate the groups with a Pseudo-Random Function (PRF) $GE : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$. For instance, the admin could generate the group identifier of e by computing $GE(e) \leftarrow LSB_b(H_s(e))$, where LSB_b gets the least significant b bits of its input. The elements with the same $GE(e)$ values are in the same groups. In this case, the number of elements in each group might be different, and the occurrences of the elements in the same group might have big differences. Note that the GE function can be modified based on the data distribution. For instance, to ensure each group contains no less than λ elements, b can be smaller. The issue is more dummy records are required than the case that groups the elements based on their occurrences. The benefit of doing so is that the users do not need to store the elements in each group, or inquire the OPS about the group identifier of each query. Moreover, since GE is deterministic, the elements never change their groups. Considering the storage on the CSP is cost-effective, in *ObliviousDB* the admin and users use GE to generate group identifiers for records and queries. However, based on the data distribution and data types in the database, the data could also be grouped in other ways. The key point is users should be able to know the group identifier of the elements in records and queries.

Database Bootstrapping. After setting up the system, the admin bootstraps the database. First, the admin populates D_f for each field f, and then generates the group identifier $GE(e)$ for each element e. For each group $GE(e)$, the admin counts the occurrences of its elements and takes the highest occurrence as its threshold τ. Finally, as described in Section 3.3.1, the admin generates dummy records to ensure the elements in each group $GE(e)$ occurs in τ records.

To ensure search efficiency, the admin also builds an index for each group $GE(e)$ by collecting the IDs of the real and dummy records containing the elements in $GE(e)$. The index allows the CSP to search only the records in the group of each query, rather than the whole database.

ObliviousDB achieves both forward and backward privacy. That is, even if the CSP holds old queries, they cannot match new records. Similarly, if the CSP holds deleted records, they cannot match new queries. To achieve both properties, our solution is to blind both the records and queries with one-off nonces. Only the records and queries blinded with the latest nonces could match. Since the records are always searched group by group, the admin generates a

Algorithm 1 $RcdEnc(rcd, flag, s)$

if $tag = 1$ **then**
  for each element $e_f \in rcd$ **do**
    $\sigma_f \leftarrow H_s(e_f)$
    $GE(e_f) \leftarrow PRF(\sigma_f)$
    $r \xleftarrow{\$} Z_p^*, SE(e_f) \leftarrow (u_1 = g_1^r, u_2 = g_1^{\sigma r})$
    $r \xleftarrow{\$} Z_p^*, DE(e_f) \leftarrow (v_1 = g_1^r, v_2 = h^r e_f)$
if $tag = 0$ **then**
  for each element $e_f \in rcd$ **do**
    $\sigma_f \leftarrow H_s(e_f)$
    $GE(e_f) \leftarrow PRF(\sigma_f)$
    $r \xleftarrow{\$} Z_p^*, SE(e_f) \leftarrow (u_1 = g_1^r, u_2 = g_1^{\sigma r})$
    $DE(e_f) \leftarrow (v_1 \xleftarrow{\$} \mathbb{G}_1, v_2 \xleftarrow{\$} \mathbb{G}_1)$
return $Ercd = ((SE(e_1), DE(e_1)), ..., (SE(e_F), DE(e_F)), tag)$ and $Grcd = ((GE(e_1), \sigma_f), ..., (GE(e_F), \sigma_f))$

nonce for each group and blinds the elements in the same group with the same nonce. After executing a query over a given group, a new nonce will be generated and used to re-blind the searched records accordingly. The technical details about the nonce blinding are given in Section 4.2.4.

In *ObliviousDB*, the group information is called *GDB*, and will be stored and managed by the OPS. Table 4.1 illustrates an example of *GDB*. Let us assume that there is a table Staff (Table 4.1(a)) containing *Name* and *Age* fields. Table 4.1(b), shows the group information of the example table. For different fields, the group information can be stored in different tables. In the shown example, we use the pair $(f, g_{i,f})$ to identify the groups in field f. Specifically, for each group, *GDB* stores a nonce, a list of identifiers of the records belonging to this group, and its elements set E. As mentioned above, the nonce is used to blind records and queries. When a user issuing a query, the OPS sends to the CSP the corresponding list of records IDs to be searched. After the searching, the OPS generates a new nonce and re-blinds the searched records. E will be retrieved by users when generating dummy records. In case the OPS is compromised, only the keyed hash value of each element is stored in E.

4.2.3 Data Encryption

After building the group information, the admin encrypts both the real and dummy records. The details of record encryption are shown in Algorithm 1 *RcdEnc* and Algorithm 2, *Nonce-Blind*.

***RcdEnc*$(rcd, flag, s) \rightarrow Ercd$.** *RcdEnc* takes the record $rcd = (e_1, ..., e_F)$, a flag tag marking if rcd is real or dummy and the secret key s as the inputs, and outputs the encrypted record

Algorithm 2 $NonceBlind(Ercd, Grcd, GDB, counter)$

1: $Ercd.ID \leftarrow counter$
2: $flags[Ercd.ID] \leftarrow Ercd.tag$
3: $counter++$
4: **for** each $GE(e_f) \in Grcd$ **do**
5: **if** $(f, GE(e_f)) \in GDB$ **then**
6: $n \leftarrow GDB(f, GE(e_f))$
7: $Ercd.SE_n(e_f) \leftarrow (u_1 \leftarrow u_1^n, u_2)$
8: **if** $H_s(e_f) \notin GDB(f, GE(e_f)).E$ **then**
9: Add $H_s(e_f)$ to $GDB(f, GE(e_f)).E$
10: **else**
11: $n \xleftarrow{\$} Z_p^*$
12: $Ercd.SE_n(e_f) \leftarrow (u_1 \leftarrow u_1^n, u_2)$
13: $GDB(f, GE(e_f)) \leftarrow (n, \{ID\}, \{H_s(e_f)\})$
14: **return** $Ercd = (ID, (SE_n(e_1), DE(e_1)), \ldots, (SE_n(e_F), DE(e_F)))$, $counter$ and GDB

$Ercd$ and the group information $Grcd$ of rcd. In particular, $tag = 1$ when rcd is real, otherwise $tag = 0$. For both real and dummy records, the admin first generates the group information for each element by running $GE(e_f)$. Each element in real records is encrypted under Searchable Encryption (SE) (Line 5) and Data Encryption (DE) (Line 6), where SE makes the encrypted data searchable, and DE ensures the confidentiality of the retrievable data. Both DE and SE are semantically secure because of the random values r, which prevents the CSP learning any frequency information in EDB. Note that for real records, only the public key h is used in DE, which means an attacker could also execute DE. To defend against this attack, we assume that there is an authentication mechanism in place to authenticate users. Only the records from authenticated users and admin will be accepted by the OPS and CSP.

To ensure the dummy records could match queries, their SE parts are generated in the same way as real records (Line 11). However, since the users do not retrieve data from dummy records, their DE parts are assigned random values (Line 12). Recall that in Section 3.1, we mentioned that the dummy records should be filtered out easily by users (R4). To meet this requirement, the OPS also maintains a list of indices that contain dummy records (not shown in Table 4.1(b)); this could be an N-bit string $flags$, where N is the total size of the database on the CSP. We have $flags[id] = 0$ or 1 if the record is a dummy or real record, respectively.

***NonceBlind*$(Ercd, Grcd, GDB, couner) \rightarrow (Ercd, counter, GDB)$.** Moreover, to ensure both the forward and backward privacy, the admin also runs *NonceBlind* to blind the SE parts. *NonceBlind* takes $Ercd$, $Grcd$, GDB and $counter$ as the inputs and returns the blinded records $Ercd$, updated $counter$ and GDB. The number $counter$ is kept by the OPS, and it is used to count the total number records stored in the CSP and assign ID to each $Ercd$ (Line 1, Algorithm 2). After determining the ID for $Ercd$, the admin assigns $flags[Ercd.ID]$ with $Ercd.tag$, such users can learn if the ID-th record stored in the CSP is real or dummy (Line 2). With $Grcd$

and *GDB*, the admin can get the nonce n for each SE part, and then blinds it with n (Line 7). Note that after bootstrapping the database, users might insert new elements that not in existing groups into the database. For a new group, a new nonce $n \xleftarrow{\$} \mathbb{Z}_p^*$ will be generated, meanwhile, its index and elements set will be built as well (Line 11 - 13).

After encrypting all the records, the admin uploads the *GDB* to the OPS, and uploads the encrypted database *EDB* to the CSP. The encrypted Staff table is illustrated in Table 4.1(c). In *EDB*, each encrypted record *Ercd* is identified with a unique ID, and each element is encrypted with both SE and DE. The last record in Table 4.1(c) is a dummy and its DE parts are random strings.

4.2.4 Query Encryption and Execution

Algorithm 3 $Query(Q,s)$

1: $user_i(Q)$:
2: $\sigma \leftarrow H_s(Q.e)$
3: $GE(Q.e) \leftarrow PRF(\sigma)$
4: $EQ.type \leftarrow Q.type$
5: $EQ.operator \leftarrow Q.operator$
6: $EQ.f \leftarrow Q.f$
7: $r \leftarrow Z_p^*, EQ.e^* \leftarrow (t_1 = g_2^r, t_2 = g_2^{\sigma r})$
8: Send EQ and $GE(Q.e)$ to the OPS

9: $OPS(EQ, GE(Q.e))$:
10: $(n, IL) \leftarrow GDB(EQ.f, GE(Q.e))$
11: $EQ.e^* \leftarrow (t_1 \leftarrow t_1^n, t_2)$
12: Send (IL, EQ, i) to the CSP, where i is the identifier of the *user*
13: Send $flags$ to the user

14: $CSP(IL, EQ, i)$:
15: $SR \leftarrow \emptyset$
16: $(t_1, t_2) \leftarrow EQ.e^*$
17: **for** each $id \in IL$ **do**
18: $\quad (u_1, u_2) \leftarrow EDB(id, EQ.f^*).SE_n$
19: $\quad$ **if** $\mathfrak{e}(u_1, t_2) = \mathfrak{e}(u_2, t_1)$ **then**
20: $\quad\quad SR \leftarrow SR \cap EDB(id)$
21: **for** each $DE(e) = (v_1 = g_1^r, v_2 = h^r d) \in SR$ **do**
22: $\quad DE'(e) \leftarrow (v_1, v_2' = v_2 * v_1^{-x_{i2}} = g_1^{x_{i1} r} e)$, where x_{i2} is the CSP side key for $user_i$
23: Send SR to $user_i$

In this scheme, we focus on the simple equality query only with one predicate, *e.g.,* 'select * from Staff where name=Alice'. The query with multiple predicates can be performed by checking each record with the predicates field by field. To support range queries, we use the same approach presented in [90]. In the following, we give the details for processing one single

equality predicate. Every query executed in *ObliviousDB* is performed with the cooperation of the user, the OPS, and the CSP. The details of the steps performed by each entity are described in Algorithm 3.

***Query*$(Q,s) \rightarrow$ *SR*.** When issuing a query $Q = (type, f, operator, e)$, it is first encrypted by the user (Lines 1-8, Algorithm 3). Specifically, the user first generates the group of $Q.e$ by computing $GE(Q.e)$. Then, it encrypts Q with K_{U_i}. Recall that we do not make effort to protect the query type, operator and the field pattern from the CSP. Thus, the user does not encrypt them, and only encrypts the interested keyword $Q.e$. Formally, $Q.e$ is randomised with SE. Since SE is semantically secure, the CSP is unable to infer the search pattern from the encrypted keyword $EQ.e^*$. Finally, the user sends both the encrypted query $EQ = (type, f, operator, e^*)$ and its group $GE(Q.e)$ to the OPS.

Given $GE(Q.e)$ and EQ, the OPS first gets its nonce n and index IL, and then blinds $EQ.e^*$ with n (Line 9 - 12). After that, the OPS sends EQ, IL, and the identifier of the $user_i$ to the CSP. Meanwhile, the OPS sends $flags$ to the user so that the user can filter out dummy records.

In the next step, the CSP checks each record in IL with $EQ.e^*$ by performing the pairing map operation (Lines 17 - 20). Assume the searched data element is $SE_n(e) = (u_1 = g_1^{r'n'}, u_2 = g_1^{\sigma' r'})$ and the encrypted keyword is $EQ.e^* = (t_1 = g_2^{rn}, t_2 = g_2^{\sigma r})$. The equality check between them is performed by checking whether $\mathfrak{e}(u_1, t_2) = \mathfrak{e}(u_2, t_1)$. Formally,

$$\mathfrak{e}(u_1,t_2) = \mathfrak{e}(u_2,t_1) \iff \mathfrak{e}(g_1^{r'n'}, g_2^{r\sigma}) = \mathfrak{e}(g_1^{r'\sigma'}, g_2^{rn}) \iff \mathfrak{e}(g_1,g_2)^{r'n'r\sigma} = \mathfrak{e}(g_1,g_2)^{rnr'\sigma'}$$

When $\sigma = \sigma'$ and $n = n'$, the equality holds, while inequality holds with negligible probability if $\sigma \neq \sigma'$ and $n \neq n'$. That is, the record rcd matches the query Q only when $Q.e = rcd.e_f$ and they are blinded with the same nonce as well. Once the matched records are found, the CSP pre-decrypts their DE parts with $user_i$'s CSP side key x_{i2} (Line 22). The search result SR is sent to $user_i$. Yang *et al.* introduce a similar method to perform the equality check for SE schemes in [91]. However, their method leaks the search pattern and frequency information of records to the CSP, since the pairing map they use is symmetric. That is, the CSP could still infer if they are the same or not by running the bilinear map operation between two records or two queries, although they are encrypted with a probabilistic algorithm.

Finally, with $flags$, the user filters the dummy records out, and decrypts the real records with her user side key x_{i1}. Specifically, the user checks if each returned record $Ercd_{id}$ is real or dummy by checking if $flags[id] = 1$. If yes, the record is real, and its $DE' = (v_1 = g^r, v_2' = g_1^{x_{i1} r})$ is decrypted with x_{i1} by computing $v_2' * v_1^{-x_{i1}} = g_1^{x_{i1} r} e * g_1^{-x_{i1} r} = e$.

Discussion. In *ObliviousDB*, if the CSP colludes with a user, they could recover the master key x. To withstand this attack, we can split the master key MSK into three shares $(K_{U_i}, K_{S_i}, K_{P_i})$, and distribute the third share K_{P_i} to the OPS. In this case, data retrieval has to be performed in

three rounds of decryption. That is, the OPS should perform the second round of decryption with K_{P_i} after filtering out the dummy records. In this case, the collusion between any two of the entities is unable to recover the master key, unless the user compromise the OPS, and colludes with the CSP.

4.2.5 Oblivious Algorithm

Algorithm 4 $Oblivious(EQ.f, GE(Q.e))$

1: $(n, IL) \leftarrow GDB(EQ.f, GE(Q.e))$
2: $Ercds \leftarrow EDB(IL)$ {Get from CSP all the records indexed by IL}
3: $n' \xleftarrow{\$} Z_p^*$, $GDB(EQ.f, GE(Q.e)).nonce \leftarrow nn'$
4: **for** each record $Ercd \in Ercds$ **do**
5: $\quad SE_n(e_{Q.f}) = (u_1 \leftarrow u_1^{n'}, u_2)$
6: $\quad$ **for** each $(SE_n(e), DE(e))$ pair in $Ercd$ **do**
7: $\quad\quad r \xleftarrow{\$} Z_p^*, SE_n(e) = (u_1 \leftarrow u_1^r, u_2 \leftarrow u_2^r)$
8: $\quad\quad r \xleftarrow{\$} Z_p^*, DE(e) = (v_1 \leftarrow v_1^r, v_2 \leftarrow v_2^r)$
9: Shuffle $Ercds$ and upload it to CSP
10: Shuffle $flags$ accordlingly
11: Update the indices of affected groups

To hide the access pattern and ensure forward and backward privacy, in the oblivious algorithm (Algorithm 4), the OPS shuffles and re-randomises all the records included in the searched group every time a query is executed.

To ensure a high level of security, we shuffle all the records in the searched group. In this case, the CSP can only recognise if two queries are performed within the same group or not. To improve the performance of the system, the number of records to be shuffled can be reduced but at the cost of security guarantees. Note that, at this stage, the user has already obtained the search results from the CSP and does not need to wait for the shuffle operation to be completed.

***Oblivious*$(EQ.f, GE(Q.e)) \rightarrow Ercds$.** The OPS first gets all the searched records from the CSP. Second, the OPS generates a new nonce and updates its *GDB*. Then, the OPS re-blinds the searched SE of each record with the new nonce (Line 5). Consequently, the queries blinded with previous nonces cannot match the re-blinded records. Similarly, a new query blinded with the latest nonce cannot match deleted records. That is, both forward and backward privacy are ensured.

In case the CSP could tell if any two queries in the same group matching the same records or not, the OPS also shuffles and re-randomises both the SE and DE parts of all the searched records. Specifically, for each element in searched records, the OPS re-randomises its SE and DE parts (Line 6 - 8). After that, the OPS shuffles all the searched records and sends them back to the CSP (Line 9). Because of the shuffling and re-randomising, if the same query is executed

again, the search result will be totally different from the previous ones in terms of the storage locations on the CSP and ciphertext. Thus, the CSP is unable to infer if different search results contain the same records or not, *i.e.,* the access pattern is protected.

Note that, when shuffling the records, $flags$ and the indices of the groups in other fields will be affected. Therefore, the OPS also needs to shuffle $flags$ and update the indices of affected groups accordingly.

4.2.6 Database Updating

Algorithm 5 $Insert(rcd, s)$

$user_i(rcd)$:
$Update \leftarrow \emptyset$, $Insert \leftarrow \emptyset$, $sign \leftarrow 0$
$(Ercd, Grcd) \leftarrow RcdEnc(rcd, 1, s)$
$Insert \leftarrow (Ercd, Grcd)$
for each $e_f \in rcd$ **do**
    $(SR_f, flags) \leftarrow Query(e_f)$
    if All the records in SR_f are real **then**
        $sign = 1$, Break;
if $sign = 0$ **then**
    for $f = 1$ to F **do**
        Assume $Ercd_{id} \in SR_f$ is dummy, $Ercd_{id}.e_f^* \leftarrow SE(NULL)$
        $Update \leftarrow Update \cup Ercd_{id}$
else
    for each $(f, GE(e_f)) \in Grcd$ **do**
        $E_{f,GE(e_f)} \leftarrow GDB(f, GE(e_f))$
    $M = \max\{|E_{f,GE(e_f)}| \mid 1 \leq f \leq F\}$
    for $m = 1$ to M **do**
        Generates $rcd_m = (NULL, ..., NULL)$
    Assign the elements in $(E_{1,GE(e_1)}, ..., E_{F,GE(e_F)})$ to $\{rcd_1, ..., rcd_M\}$
    for $m = 1$ to M **do**
        $(Ercd, grcd) \leftarrow RcdEnc(rcd_m, 0, s)$
        $Insert \leftarrow Insert \cup (Ercd, Grcd)$
Send $Insert$ and $Update$ to the OPS

$OPS(Insert, Update, GDB)$:
for each $(Ercd, Grcd) \in Insert$ **do**
    $Ercd \leftarrow NonceBlind(Ercd, Grcd, GDB, counter)$
    Remove $Grcd$ from $Insert$
Send $Insert$ and $Update$ to the CSP

$CSP(Insert, Update, EDB)$:
for each $Ercd \in Insert$ **do**
    Insert $Ercd$ into EDB
for each $Ercd \in Update$ **do**
    Update $EDB(Ercd.ID)$ with $Ercd$

To ensure the CSP cannot infer the search pattern of the queries in the same group from the size pattern, the elements in the same group should have the same occurrence, which has been achieved by inserting dummy records when bootstrapping the database. In this work, we consider dynamic databases, where users can still insert, delete and update records if required after bootstrapping the initial database. However, inserting, deleting, or updating records directly to/from the database will change the occurrence of each involved element. In Chapter 3, we have discussed the methods to ensure the elements in the same group have the same occurrence when inserting, deleting, and updating records. In this section, we take the insert query as an example and give more technical details. Note that if the database is grouped based on the elements occurrences, *ObliviousDB* could also use *Solution 2* given in Section 3.3.2 to manage the dummy records when executing insert and delete queries.

The details of inserting a record are illustrated in Algorithm 5. Basically, the records are inserted with the cooperation of the user, OPS, and CSP.

***Insert*$(rcd, s) \rightarrow$ (*Insert*, *Update*).** The user first encrypts the record $rcd = (e_1, ..., e_F)$ by running *RcdEnc* (Line 3). When inserting rcd, as discussed in Section 3.3.1, to ensure the elements in each involved group $(f, GE(e_f))$ have the same occurrence, the user could either ensure the occurrences of each inserted element e_f unchanged by setting one of the dummy records with e_f to 'NULL' in field f, or increase the occurrence of other elements in group $GE(e_f)$ also by one by inserting dummy records. In *ObliviousDB*, we use the former solution since it requires much fewer dummy records than the latter one. However, if there is no dummy record containing each inserted element, the user can only insert dummy records to increase the occurrence of other elements in involved groups. Therefore, on the user side, the second step is to check if there is a dummy record containing each inserted element (Line 2 - 8). Specifically, the user first initialises two sets $Update$ and $Insert$ to hold the records to be updated and the records to be inserted, respectively. Indeed, only one of them will be used, *i.e.,* $Update$ will be used in the former case and $Insert$ will be used in the latter case. Second, the user takes each inserted element e_f as the interested keyword and gets SR_f and $flags$ (Line 6). Then, the user checks if there is a dummy record in each SR_f (Line 7 - 8). If yes, in the next step, the user updates the f field of one dummy record to $SE(NULL)$ (Line 12). Otherwise, the user generates and encrypts dummy records (Line 14 - 22). Specifically, in the latter case, for each field f the user first gets the elements set $E_{f,GE(e_f)}$ of group $(f, GE(e_f))$ from the OPS (Line 15), which is stored in *GDB*. Assume there are $|E_{f,GE(e_f)}|$ distinct values in group $(f, GE(e_f))$. Let M be the maximum of all these numbers, *i.e.,*

$$M = \max\{|E_{f,GE(e_f)}| \mid 1 \leq f \leq F\}$$

meaning the involved biggest group contains M elements. The user generates M records with

'NULL' values. Second, it assigns the elements in $E_{f,GE(e_f)}$ to the M records randomly. Third, the user encrypts each dummy record with *RcdEnc* (Line 21). Finally, the user sends *Insert* and $Update$ to the OPS.

The OPS blinds all the records in *Insert* with nonces by running *NonceBlind* (Line 24 - 28). For the records in $Update$, the OPS does not blind the $SE(NULL)$ values since we do not need to ensure its forward and backward privacy. Finally, the OPS sends *Insert* and $Update$ to the CSP.

The CSP inserts all the records in *Insert* into EDB and updates the records in $Update$ (Line 29 - 33). Finally, the OPS and the CSP run the *Oblivious* function to shuffle and re-randomise all the records in involved groups.

4.3 Security Analysis

In this section, we prove *ObliviousDB* protects the search, size, and access patterns, and achieves the forward and backward privacy from an honest-but-curious CSP.

Leakage. Although *ObliviousDB* leaks much less information than other schemes, it still suffers $\mathcal{L}0$ leakage defined in Section 2.2. Specifically, the CSP can learn if users are inserting, deleting, updating, or selecting records, and if two queries are searched over the same fields or not since we do not re-randomise the fields and shuffle the records column by column in the oblivious algorithm. Moreover, for select queries, the CSP can also learn if they are equality or range queries. In addition, *ObliviousDB* also leaks the group information to the CSP. To optimise the performance of the system, *ObliviousDB* divides the data into groups and only performs the search and oblivious operations within groups. Consequently, the CSP could learn if some records and searched keywords are in the same groups or not. Note that, by hiding the grouping method from the CSP, it cannot infer the relationship between the elements in plaintext. Given these leakages, we give our security definition as below. Specifically, we only consider the queries with the *same structure*.

Definition 7 (The same structure). We say any two queries Q_0 and Q_1 have the same structure if they satisfy the following:

- $Q_0.type = Q_1.type$, *i.e.,* they have the same type. Herein, since the database is dynamic, the query type can be insert, select, or delete. $Q = rcd$ for an insert query.
- $Q_0.operator = Q_1.operator$. In *ObliviousDB*, we focus on equality queries.
- $Q_0.f = Q_1.f$, *i.e.,* they are searching over the same field.
- $GE(Q_0.e) = GE(Q_1.e)$, *i.e.,* they are in the same group. For the insert queries in the same structure, the new element for each field should be in the same group.

For relational databases, it is hard to hide the query type and operator from the CSP since all the operations over the outsourced database are performed by the CSP. One possible way to hide the query type is reading and writing the encrypted database for each query. Similarly, the query operator can be hidden by always performing one type of operator for all queries and let the user do further process locally. However, this will add the computation burden on the user side. As mentioned, the interested field can be hidden by shuffling the database column by column after each query, and the group information can be protected by padding all the elements into the same occurrence and searching the whole database for executing each query. Unfortunately, all these operations will significantly impact the performance of the system. In fact, comparing with the search, size, and access patterns, the above four properties leak much less information about the query and records to the CSP. For instance, if the query is a range query, the CSP can only learn the elements in the searched field are integers. However, this information does not reveal much about this field since the integer could represent many attributes, such as age, date, price, and temperature. Indeed, whether the elements are integers or strings can also be learned from the length of their ciphertext if they are not padded before being encrypted. On the other hand, even with the above restrictions, as shown in the following definition, the adversary can still design the queries with different search, size, and access patterns by assigning different values to $Q.e$. If these patterns are not protected in *ObliviousDB*, the adversary can win the game by mounting the leakage-based attacks, such as the count attack, IKK attack, and file/record-injection attack.

Security definition and analysis. The CSP is modelled as the Probabilistic Polynomial-Time (PPT) honest-but-curious adversary $\mathcal{A}$, which means $\mathcal{A}$ honestly follows the designated protocols and gets all the messages the CSP sees, but it never mounts active attacks, such as modifying the database and messages. The scheme is considered to be secure if an adversary could break it with not more than a negligible probability. Formally, it could be defined as follows:

Definition 8 (**Negligible Function**). A function f is negligible if for every polynomial $p(.)$ there exists an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.

Definition 9. Let $\prod_{ObliviousDB}$ =(Setup, Query, Oblivious) be *ObliviousDB*, and λ and k be the security parameters. $\mathcal{A}$ is a Probabilistic Polynomial-Time (PPT) adversary, and $\mathcal{C}$ is a challenger. The game between $\mathcal{A}$ and $\mathcal{C}$ in $\prod_{ObliviousDB}$ is described as below:

- **Setup** The challenger $\mathcal{C}$ first initialises the system by generating *Params* and MSK. Then, she generates the secret key pair (K_U, K_S). The adversary $\mathcal{A}$ is given *Params* and K_S.
- **Bootstrap** $\mathcal{A}$ submits a database Δ[1]. As done by the Admin, $\mathcal{C}$ encrypts Δ and divides

[1]For simplicity, we assume there is only one single table in Δ and regard Δ as a table. Without loss of generality, our proofs will hold for a database containing a set of tables.

the data in each field into groups. Moreover, $\mathcal{C}$ generates a number of dummy records for each group, such that the elements in each group have the same occurrences. The encrypted database EDB is sent to $\mathcal{A}$. The encrypted group information GDB is securely kept by $\mathcal{C}$.

- **Phase 1** $\mathcal{A}$ can make polynomially many queries Q in plaintext. $\mathcal{C}$ encrypts and blinds each query Q to an encrypted query EQ, and gets the indices list IL, as would be done by the user and the OPS. With EQ and IL, $\mathcal{A}$ searches over EDB to get the search result SR for select queries. After that, $\mathcal{C}$ and $\mathcal{A}$ engage in the oblivious algorithm to update EDB. So, for each query, $\mathcal{A}$ sees EQ, IL, SR, and the records set $Ercds$ output by the oblivious algorithm. Note that, the $\mathcal{A}$ could cache EQ and execute it again independently at any time.

- **Challenge** $\mathcal{A}$ sends two queries Q_0 and Q_1 to $\mathcal{C}$ that have the same structure, which can be those already issued in phase 1. $\mathcal{C}$ responds the request as follows: it chooses a random bit $b \in \{0,1\}$ and encrypts and blinds Q_b, as done by the user and OPS, to EQ and IL. Then, $\mathcal{C}$ and $\mathcal{A}$ perform the full protocol, so that $\mathcal{A}$ learns SR and $Ercds$.

- **Phase 2** $\mathcal{A}$ continues to adaptively request polynomially many queries, which could include the challenged queries Q_0 and Q_1 and insert or delete records.

- **Guess** $\mathcal{A}$ submits her guess b'.

The advantage of $\mathcal{A}$ in this game is defined as:

$$Adv_{\mathcal{A},\prod_{ObliviousDB}}(1^k) = Pr[b' = b] - \frac{1}{2}.$$

We say *ObliviousDB* achieves the privacy of the search, size, and access pattern, and forward and backward privacy, if all PPT adversaries have a negligible advantage in the above game.

In this game, $\mathcal{A}$ is powerful. She knows the plaintext of all the real records and queries and has full access to EDB. So she could learn the real search result of Q_0 and Q_1 and run all the encrypted queries over EDB to get the encrypted search results at any time. If one of the search, access, size, and forward and backward privacy is not protected, $\mathcal{A}$ could infer b easily. For example, if the search pattern is not protected, $\mathcal{A}$ could select one of the queries issued in phase 1 as either Q_0 or Q_1, and win the game by comparing EQ with previous encrypted queries. If the forward and backward privacy is not guaranteed, $\mathcal{A}$ could delete and insert some records in phase 1, and search for the deleted or inserted element either in Q_0 or Q_1. Similarly, $\mathcal{A}$ could win the game by setting other special Q_0 or Q_1 if one of the other properties is not achieved. However, if under this setting $\mathcal{A}$ cannot win the game with non-negligible advantage, it means *ObliviousDB* achieves all the properties. Moreover, in the above definition, $\mathcal{A}$ can actively

insert malicious data, and update or delete users' data in Phase 1 and 2. As far as we know, we are the first to consider an active adversary in the security model setting. In the security proof of most previous SE schemes, the adversary only passively receives encrypted queries and database. Since these schemes suffer from leakage, they only claim the leakage profile but ignore the ability of the adversary to analyse the leakage and inject malicious data.

Theorem 1. Let the SE and DE have semantic security. Then *ObliviousDB* achieves the privacy of the search, size, and access pattern, and forward and backward privacy.

Proof. (Sketch) We show that the bit b chosen by $\mathcal{C}$ is information-theoretically hidden from the view of $\mathcal{A}$, assuming that both SE and DE are semantically secure.

Static database. Herein, we first consider a static database and focus on select queries. Consider the view of $\mathcal{A}$ in the game. $\mathcal{A}$ chooses an arbitrary database Δ and uploads this to $\mathcal{C}$. In Phase 1, $\mathcal{A}$ makes queries that are answered correctly by $\mathcal{C}$ by following the protocols.

In the challenge round, $\mathcal{A}$ sends two queries Q_0 and Q_1. $\mathcal{A}$ receives EQ. By definition, the two queries have the same structure. Hence, the same type, interested field, operator and indices list *IL*, will be received by $\mathcal{A}$ for either query. Since *SE* is semantically secure, $\mathcal{A}$ cannot distinguish the query terms given the ciphertexts $EQ.e^*$.

The involved group is accompanied by a nonce n. With overwhelming probability, n is distinct and unrelated to the nonces used in previous queries. Previously encrypted search keywords can no longer be used to query these indices, and EQ cannot be executed over deleted records since the nonces do not match. Hence, there is no way to link information from previous search queries to these records, indicating forward and backward privacy is achieved.

The adversary $\mathcal{A}$ also gets the search result *SR*. Although the numbers of real records matched with Q_0 and Q_1 are known to $\mathcal{A}$ since all the queries and real records in plaintext are set by her, a number of dummy records are inserted into EDB to ensure the queries in the same group always match the same number of records. That is, no matter what b is, $|SR|$ is fixed. Since SE and DE are semantically secure, the dummy records are indistinguishable from the real ones. Therefore, the CSP is unable to distinguish the two queries from *SR*, indicating the privacy of size pattern is achieved.

Even if the queries Q_0 and/or Q_1 have previously been executed by $\mathcal{A}$, the shuffling and re-randomising performed by the oblivious algorithm, imply that $\mathcal{A}$ cannot distinguish the two queries by comparing *SR* with previous search results, indicating the access pattern privacy is achieved.

Finally, $\mathcal{A}$ gets $Ercds$ from the oblivious algorithm. All the records and re-randomised. Due to the semantic security of *SE* and *DE*, $Ercds$ leaks nothing meaningful to $\mathcal{A}$.

The game continues in Phase 2. $\mathcal{A}$ may repeat Q_0 and/or Q_1. $\mathcal{A}$ may run a related select query to test the search result. Again, due to the shuffling and re-randomising operations, the

ciphertext and storage locations of the matched records for Q_0 and/or Q_1 will be different from *SR*. Similarly, the nonce updating does not allow records to be linked to records found in previous search queries. Hence, the future state of the database and the queries in Phase 2 are independent of the query made in the challenge round.

Since $\mathcal{A}$ has no information to distinguish the bit b, the scheme satisfies the definition.

Dynamic database. *ObliviousDB* also supports the insert and delete queries over dynamic databases. In this part, we analyse the security of *ObliviousDB* when involving delete and insert queries.

If Q is a record to be inserted, for each element e_f included in Q, $\mathcal{C}$ first selects all the records containing e_f and checks if one of them is dummy. If yes for the F elements, $\mathcal{C}$ updates the F dummy records and set their respective fields to 'NULL'. Otherwise, $\mathcal{C}$ increases the occurrences of the elements in the F involved groups by inserting dummy records. In both cases, all the elements in the F involved groups will still have the same occurrence. After that, all the records in the F involved groups will be shuffled and re-randomised. During this process, $\mathcal{A}$ searches over *EDB*, inserts new records into *EDB*, and engages in the oblivious algorithm. Thus, in addition to EQ, IL, SR, and $Ercds$ involved in the searching and shuffling operations, $\mathcal{A}$ also gets the encrypted real and dummy records to be inserted.

If Q is a delete query, $\mathcal{C}$ first selects the matched records and then sets the real ones to dummy by changing the $Flags$, which is unknown to $\mathcal{A}$. After that, the records in the involved groups will be re-randomised and shuffled. In this process, $\mathcal{C}$ also gets EQ, IL, SR, and $Ercds$ for the searching and shuffling operations.

First, we analyse if $\mathcal{A}$ can win the game by inserting or deleting records in phase 1. For both the insert and delete queries, *ObliviousDB* always ensures the elements in the same group have the same occurrences. As long as Q_0 and Q_1 are in the same group, $|SR|$ is independent of b no matter what $\mathcal{A}$ did in phase 1. Moreover, due to the SE and DE encryption, $\mathcal{A}$ is unable to infer b based on the ciphertext of the records.

Second, we discuss if $\mathcal{A}$ can win the game when Q_0 and Q_1 are two insert or delete queries in the challenge phase. When inserting a record, $\mathcal{A}$ gets $EQ, IL, SR, Ercds$, and the encrypted real and dummy records to be inserted. For delete queries, $\mathcal{A}$ only gets EQ, IL, SR, and $Ercds$. As discussed above, the value of b is independent from EQ, IL, SR, and $Ercds$. Due to the semantic security of both the SE and DE encryption, $\mathcal{A}$ cannot tell the value of b based on the encrypted records. □

4.4 Performance Analysis

We implemented *ObliviousDB* in C using the MIRACL 7.0.0 library, necessary for cryptographic primitives. The implementation of the overall system, including the functions on the

Table 4.2: The storage overhead with different numbers of groups.

#Groups	#Dummy records	#Elements in each group	#Records in each group
1	2599836	99996	=4099836
10	2389842	10000	≈38000
100	1978864	1000	≈35000
1000	1567983	100	≈3000
10000	1034007	10	≈240

user, the OPS, and the CSP, was tested on a single machine with Intel *i*5 3.3 GHz processor and 8 GB RAM running Ubuntu 14.08 Linux system. In our testing scenario, we ignored the network latency that could occur in a real deployment. In the following, all the results are averaged over 10 trials.

We evaluated the performance using TPC-H benchmark [86]. The table used in our experiment was the 'ORDERS', which consists of 1.5 million records and 9 fields. More details about the 'ORDERS' table can be found in Section 3.2. For the performance evaluation, we tested the simply equality queries like 'select * from ORDERS where O_CUSTKEY=?'. Due to the memory limitation, we only encrypted the elements in field 'O_CUSTKEY' with SE for searching and encrypted each record as a whole with DE encryption for data retrieval. Moreover, we divided the records into groups based on the 'O_CUSTKEY' elements. In the following, we show the performance of our scheme when changing the number of groups and the result size.

4.4.1 Group Generation

In 'ORDERS' table, all the 'O_CUSTKEY' elements are integers. For simplicity, we divided the records into groups by computing $GE(e) \leftarrow e \bmod b$ for each element e in 'O_CUSTKEY' field. Specifically, we divide the records into 1, 10, 100, 1000, and 10000 groups by setting b =1, 10, 100, 1000, and 10000, respectively.

Recall that the search and access patterns of the queries in the same groups are protected from the CSP in *ObliviousDB* since all the elements in the same group are padded into the same occurrence with dummy records and all the records in the searched group are shuffled and re-randomised after executing each query. Herein, we use the number of elements in each group to represent the security level of *ObliviousDB*. Both the number of required dummy records and the security level of *ObliviousDB* vary with the number of groups. In Table 4.2, we show the number of required dummy records, the number of elements in each group and the number of records in each group when dividing the records into 1, 10, 100, 1000, and 10000 groups. In particular, when the group number is 1, *i.e.,* all the records are in the same group, 2599836 dummy records in total are required to pad all the elements in 'O_CUSTKEY' into the same occurrence. In this case, all the 99996 different 'O_CUSTKEY's have the same occurrence and

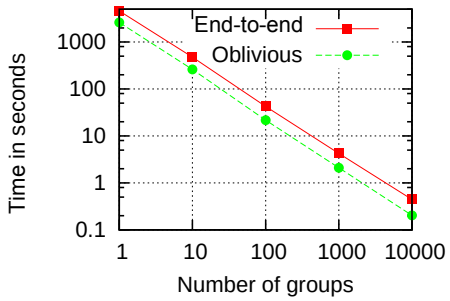


Fig. 4.2. The process time taken by each phase when changing the number of groups.

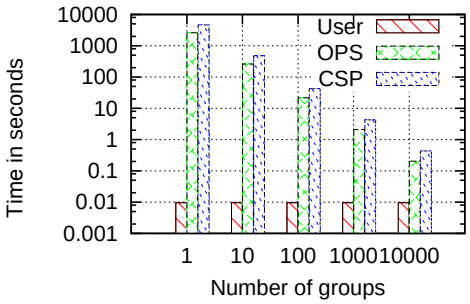


Fig. 4.3. The process time taken by each entity when changing the number of groups.

the scheme ensures the highest level of security. That is, all the queries involving the 99996 'O_CUSTKEY's match the same number of records, and the CSP cannot tell if any two of them are searching for the same records or not and if they match the same records or not. However, in this case, the CSP has to search the most number of records, which is 4099836 (1500000 real records and 2599836 dummy records). When we divide the records into more groups, as shown in Table 4.2, fewer dummy records are required, fewer records will be searched by the CSP, and fewer elements will be contained in each group. When there are 10000 groups, only 1034007 dummy records are required totally, and the CSP just needs to search around 240 records for each query. However, each group only contains 10 different elements, making it easier for the CSP to infer the search and access patterns.

4.4.2 Select Query Evaluation

To show the performance of *ObliviousDB* when dividing the database into 1, 10, 100, 1000, and 10000 groups, we tested a simple equality query over field 'O_CUSTKEY' and measured the query process time taken by each phase and entity. When dividing the records into 1, 10, 100, 1000, and 10000 groups, the searched groups contains 4099836, 410000, 34000, 3300, and 320 records, and their thresholds are 41, 41, 34, 33, and 32, respectively. Moreover, the tested query matches 32 real records in total.

Fig. 4.2 shows the query search time and the oblivious time (in logarithmic scale). The query search time consists of the query encryption time on the user, the query blind time on the OPS, the search and result pre-decryption time on the CSP, and the result decryption time on the user. From Fig. 4.2 we can see that both the query search time and the oblivious time reduces linearly with the increase of group numbers. When there are 10000 groups, the search and oblivious phases can be completed in 0.44 and 0.2 seconds (s), respectively.

Moreover, we show the computation overhead on each entity for processing a query in

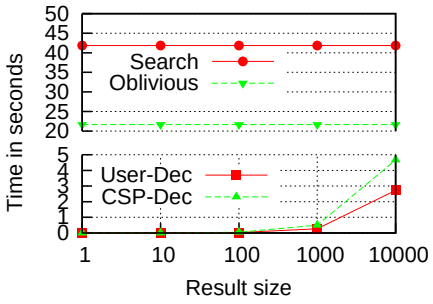


Fig. 4.4. The process time taken by each phase when changing the result size.

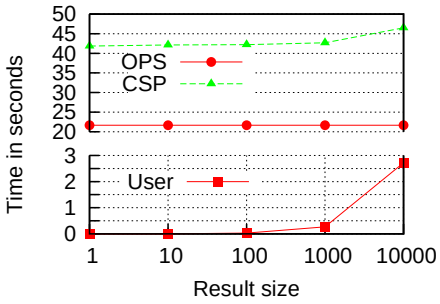


Fig. 4.5. The process time taken by each entity when changing the result size.

Fig. 4.3. As described in Section 4.2, when issuing a query, the user generates *GE* of the interested keyword, encrypts the query and decrypts the search result. The main computation overhead in *ObliviousDB* is outsourced to the OPS and the CSP. Basically, the OPS is responsible for fetching the list of indices of the searched group, blinding the query with the corresponding nonce, and shuffling and re-randomising all the searched records. The CSP searches the records and decrypts the matching records. Fig. 4.3 shows that the process time on the user is not affected by the group number, which is less than 0.01s and negligible compared with the process times on the OPS and the CSP. Note that the y-axis in Fig 4.3 is in logarithmic scale. On the contrary, the process times of both the OPS and the CSP reduce linearly with the increase of group numbers. Moreover, the computation overhead on the CSP is higher than that on the OPS.

In the second test, we fixed the group number to 100 and measured the times taken by each phase and entity when executing the queries matching 1, 10, 100, 1000, and 10000 records (including both the real and dummy records). The times taken by each phase is shown in Fig. 4.4. Specifically, we divided the query process into four phases: searching the records (Search), pre-decrypting the result on the CSP (CSP-Dec), decrypting the result on the user (User-Dec), and running the oblivious algorithm (Oblivious). From Fig. 4.4, first we can see that the times taken by the 'CSP-Dec' and 'User-Dec' phases increase with the result size. Moreover, the time taken by 'CSP-Dec' is higher than that of 'User-Dec'. The reason is that the user only decrypts real records, whereas, the CSP decrypts both the matching real and dummy records. In contrast, the performance of 'Search' and 'Oblivious' phases are not affected by the result size.

Fig 4.5 shows the times taken by the user, the OPS, and the CSP when changing the result size. Due to the result decryption operations, the times taken by the user and the CSP increases with the result size. However, the performance of the oblivious algorithm is determined by the group size, rather than the result size. Thus, the time taken by the OPS is not changed when fixing the group size and changing the result size.

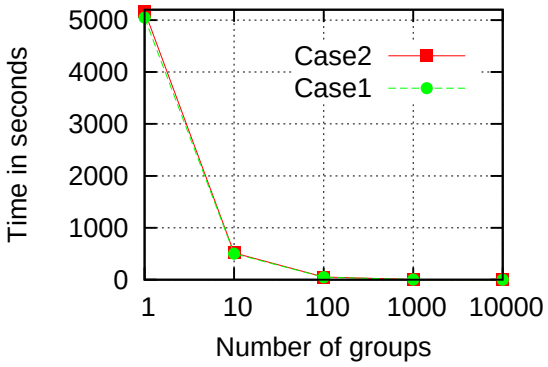


Fig. 4.6. The performance of insert queries.

4.4.3 Insert Query Evaluation

For dynamic operations, we also evaluate the performance of insert queries when dividing the database into 1, 10, 100, 1000, and 10000 groups. *ObliviousDB* leverages *Solution 1* given in Section 3.3.1 to manage the dummy records when inserting or deleting records. As described in Algorithm 5, when inserting a real record $rcd = (e_1, ..., e_F)$, the user will ensure the elements in the same group always have the same occurrences. Based on the data distribution, the insert query will be processed in two different ways. Specifically, if each $e_f \in rcd$ is included in a dummy record in the database, the user will replace e_f with 'NULL' in the dummy record. By doing so, once rcd is inserted, the occurrence of e_f is unchanged. On the contrary, if one element in rcd only included in real records in the database, its occurrence will increase by one after inserting rcd. To ensure all the elements in the same group always have the same occurrence, the user will insert a number of dummy records to ensure the occurrence of other elements in $(f, GE(e_f))$ also increase by one. In this experiment, we tested the performance of the insert query in both cases and showed the result in Fig. 4.6. In the figure, the former case is called 'Case1' and the latter is called 'Case2'.

Fig. 5 shows that the execution times of both cases decline when increasing the number of groups. The reason is that, in both cases, the execution time is dominated by the searching operation, and the searching time declines with the number of groups. Moreover, the performance of 'Case1' is slightly better than 'Case2' when there are less than 10 groups. However, the performance of the two cases is almost the same when there are more than 10 groups. That is because the performance of 'Case2' is also affected by the number of dummy records should be inserted, M. Furthermore, the value of M is determined by the number of elements in each involved group. When the database is divided into more groups, the fewer elements will be included in each group and fewer dummy records are required in 'Case2'. In this experiment, when there are 1, 10, 100, 1000, and 10000 groups, the user has to generate and encrypt 99996,

10000, 1000, 100, and 10 dummy records in 'Case2', respectively. Specifically, *ObliviousDB* takes about 1.25 milliseconds for inserting a dummy record, meaning 'Case2' takes $12.5 \times M$ milliseconds for inserting M dummy records. In contrast, in 'Case1', the user just need to update one field of F dummy records no matter how many groups there are.

Note that, in Fig. 4.6, we do not include the time for running the Oblivious algorithm. However, it is necessary to hide the access pattern.

For deleting a record, the user just needs to update its $flag$ stored in the OPS to 0.

4.5 Related Work

Since the seminal paper by Song *et al.* [3], many SE schemes have been proposed and research in this area has been extended in several directions. In this section, we categorise the approaches presented in the literature based on information leakage and key management and summarise their limitations.

4.5.1 Schemes with Controlled Leakage

Only several recent works tried to partially address the issue of information leakage. In [16], Naveed *et al.* introduce a solution called *blind storage* that protects the size pattern. The basic idea is to divide each file into a set of blocks. When a file is requested, a larger set of blocks will be downloaded and decrypted by the user. In this way, the real number of blocks of the matched file is protected. However, it aggravates the computational and storage overheads on the user side. Moreover, it fails to protect the search and access patterns, since the same query always requests the same block set.

In [92], Wang *et al.* extend *blind storage* and also protects the search and access patterns from the CSP. Specifically, for each required data item the user also downloads a set of random blocks in addition to the required ones, such the real accessed blocks are masked by the random ones. Moreover, all the downloaded blocks are permuted, re-randomised and written back to the CSP. As a result, when the same data item being accessed again, a different set of blocks will be downloaded. Thus, the CSP cannot infer the search pattern from the access pattern. However, in this scheme, the user has to store the location information for each file locally and updates them after each access, which makes the scheme cumbersome when supporting multiple users access. Moreover, it increases the storage and computation overhead on the user side.

Samanthula *et al.* [42] present a query processing framework that supports complex queries. A homomorphic encryption algorithm is used to encrypt the data in their scheme. Thus, it supports more complex queries when compared to other schemes, and conceal the size, search,

and access patterns from the CSP. However, this scheme is single user and does not scale well for databases with a large number of attributes.

Cao *et al.* [5] design a scheme that supports a multi-keyword ranked search. The scheme protects the search pattern by hiding the trapdoor linkability. Wang *et al.* [43] propose a public multi-keyword searchable encryption scheme based on Paillier [39], which also hides the size, search, and access patterns. More recently, in [41], Ishai *et al.* protect both the search and access patterns combining a PIR technique with a B-tree data structure. Although these three schemes provide different index structures for speeding up the search, the constructions are static and do not support insert, update, and delete operations.

In [19], Stefanov *et al.* design a dynamic sub-linear searchable construction based on an ORAM-like hierarchical structure and achieve forward privacy. Similarly, Rizomiliotis *et al.* [61] propose another dynamic ORAM-based scheme that achieves forward privacy and sub-linear search. More recently, the dynamic SE scheme introduced by Bost [32] also achieves forward privacy. Instead of using an ORAM-like structure, this scheme relies on a trapdoor permutation. However, it only ensures forward privacy until a new query is issued. A CSP could still learn if the new file contains the keywords searched previously, by comparing the access pattern of a new query with those of previous queries. Moreover, all these three schemes fail to ensure backward privacy.

Arx [66], introduced by Poddar *et al.*, is a database system that protects the data frequency information. Its main idea is making all the elements different by appending them with unique numbers and encrypting them with DET primitives. For issuing queries, the interested element e will also be appended with the same numbers that used in the database for e and encrypted with DET. As a result, all the encrypted elements have different ciphertext, and the CSP can perform the search efficiently. However, the user has to store the state for each distinct value in the database locally, which is not suitable for MwMr applications. Moreover, it leaks search, access, and size patterns.

In [56], Chen *et al.* introduce an SSE scheme that obfuscates the access pattern by adopting $d-$privacy [93] technique. The basic idea is to divide each file into a set of shards, from several of which the file can be recovered. The queries searching for the same file could match different sets of shards. Thus the access pattern is obfuscated. However, this solution cannot defend the file-injection attack. The problem is if some of the returned shard is injected by the CSP, it is still possible to recover queries.

4.5.2 Multi-user SE Schemes

Several works have concentrated on supporting multi-user access and simplifying key management. Curtmola *et al.* [8] introduce a Multi-User (MU) scheme by combining a single user SE scheme with a broadcast encryption scheme, where only the authorised user can issue queries

with the key received from the data owner. However, each time a user is revoked, the data owner has to generate a new key. Even worse, the data stored on the CSP is encrypted with the key shared among all the users, which means the revoked users can still recover all the data if they collude with the CSP. The MU SE scheme given by Jarecki *et al.* [54] has the same problem. That is, the data security against revoked users is achieved based on the assumption that there is no collusion between the CSP and revoked users; otherwise, the key has to be updated and the data has to be re-encrypted with the new key. Moreover, in their scheme, the data owner has to be online to generate search tokens for all the authorised users.

Hang *et al.* [12] and Ferretti *et al.* [9] present two different collusion-resistant mechanisms that support multi-user access to the outsourced data. Although they support approaches to avoid key sharing among users, in both, after user revocation, it is necessary to generate a new key and re-encrypt the data.

CryptDB [17] is a multi-user scheme where each user has her own password, which is managed by a proxy between the user and the database server. Sarfraz *et al.* [18] revisit CrtypDB and also design a multi-user scheme with a fine-grained access control mechanism. Instead of assigning the keys to users, both [17] and [18] store them in a proxy. Since the users never know the underlying encryption key, they do not require to refresh the key when revoking a user. The problem is that these two mechanisms require the proxy to be online for performing operations on behalf of the users. As a result, the proxy represents a single point of failure: an attacker who compromises the proxy will gain access to all the logged-in users' keys and data.

Sun *et al.* [24] utilise a Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [94] mechanism to achieve a scalable SE scheme that supports multi-user read and write operations without sharing any key. However, for user revocation, the data has to be re-encrypted with a new access structure and secret keys of all the other users need to be updated with a new attribute set. Strictly speaking, this scheme is also a single-user scheme.

In the literature, only the proxy-based encryption schemes, such as [4, 22, 57–60], can support multi-user access, where each user has her own key and does not require any re-encryption when an authorised user is revoked.

Many other works also investigated approaches to increase search efficiency [8, 11, 14], or data integrity and reliability in the setting where the CSP is totally untrusted [95, 96].

4.6 Conclusions and Future Work

In this work, we propose *ObliviousDB*, a searchable scheme for hybrid outsourced databases. ObliviousDB is a multi-user scheme that does not leak information about the search, access, and size patterns within each group. It is also a scheme that achieves both forward and backward privacy, where the CSP cannot reuse cached queries for checking if new records have

been inserted or if records have been deleted. We have implemented *ObliviousDB* and shown that it is capable of performing queries over a database of 4099836 records in around 4 seconds when dividing the database into 1000 groups.

Chapter 5

Multi-Cloud Based Solution

In the previous chapter, we have shown the security and performance of *ObliviousDB*. However, *ObliviousDB* also suffers from several limitations. To some extent, the security and efficiency of *ObliviousDB* rely on the trusted private cloud, which makes this scheme infeasible for small and medium-sized organisations that do not have resources for deploying. Specifically, we cannot move the functionality on the OPS to the public cloud or to users, since it will either leak sensitive information, such as if each record is dummy or real, to the CSP, or increase the computation and storage overheads on the user side. Moreover, it is hard to synchronise the storage on all the users for MwMr databases if we move the functionality of the OPS to users. The other limitation is that, in *ObliviousDB*, the search operation between encrypted queries and records is performed with the bilinear map, making the scheme computationally intensive for large databases. As shown in Section 4.4, when there are half a million records in a group, it takes about 76.7 minutes to get the search result.

In this chapter, we present a dynamic SSE scheme for multi-cloud environments named Privacy-preserving Multi-cloud Database (*P-McDb*) that addresses the two limitations. First of all, *P-McDb* does not rely on any trusted entity to manage and process any sensitive data. This means that it can be deployed in untrusted environments. Second, *P-McDb* only uses symmetric encryption and achieves much better performance than *ObliviousDB* and many other SE schemes. Furthermore, *P-McDb* can effectively resist the leakage-based attacks, such as the frequency analysis, IKK, count, file/record-injection, and reconstruction attacks.

According to the latest report given by Rightscale, 81% of enterprises already have a multi-cloud strategy in place, which means the schemes based on multi-cloud are feasible for most organisations. In the research literature, using the multi-cloud strategy is also not new. For instance, in [97] Stefanov *et al.* present a 2-cloud ORAM system that reduces the bandwidth cost between the user and the cloud. In [55], Bösch *et al.* introduce an SE scheme that hides the search pattern by distributing the search operation across two non-colluding clouds. In [64],

Hoang *et al.* propose a dynamic SSE scheme that preserves the privacy of the search and access patterns by creating a distributed encrypted incidence matrix on two non-colluding servers.

P-McDb is a dynamic SSE scheme built on two non-colluding CSPs. Comparing with the existing multi-cloud based solutions, *P-McDb* not only hides the search, access, and size patterns, but also ensures the forward and backward privacy. Moreover, *P-McDb* achieves a much better performance when compared with equivalent systems. In addition, *P-McDb* is MwMr scheme that supports a scalable key management approach. Specifically, in *P-McDb*, many users have access to the same database, and each user is able to protect her queries and search results against all the other entities. When one user is revoked, instead of changing the key and re-encrypting the data, *P-McDb* only needs to inform all the CSPs to stop any service for the revoked user. Although the revoked users still own the key, they are unable to issue queries or recover records even by colluding with one CSP. Our key technique is to use two CSPs, that are assumed not to collude: one CSP stores the data and performs the search operation, the other manages the re-randomising and shuffling of the database. A user with access to both CSPs can perform an encrypted search without leaking the search pattern, access pattern, or size pattern, while achieving forward and backward privacy.

The contributions of this chapter can be summarised as follows:

1. We propose a dynamic SSE scheme that protects the search, access, and size pattern, thus ensuring resistance against leakage-based attacks.

2. *P-McDb* achieves both forward and backward privacy.

3. *P-McDb* supports flexible multi-user access in a way that the issued queries and search results of one user are protected from the other entities. Moreover, revoking users does not require key regeneration and data re-encryption even in case if one of the CSPs colludes with the revoked users.

4. To show the feasibility of our approach, we have implemented *P-McDb* and measured its performance.

The rest of this chapter is organised as follows. In Section 5.1, we provide an overview of our approach. Solution and construction details can be found in Section 5.2. In Section 5.3, we analyse the security of our system. Section 5.4 reports the performance. We analyse the potential limitations of our scheme and give a possible solution in Section 5.5. We review the related work in Section 5.6. Finally, we conclude this chapter in Section 5.7.

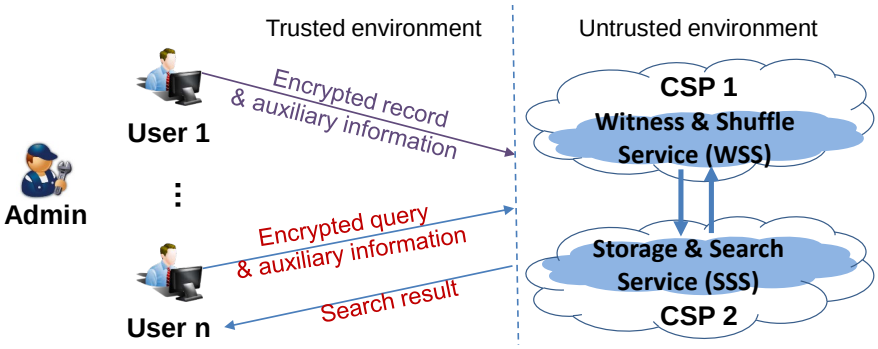


Fig. 5.1. An overview of *P-McDb*.
Users can upload records and issue queries. The SSS and the WSS represent independent CSPs. The SSS stores encrypted records and executes queries. The WSS stores auxiliary information to ensure privacy and provides auxiliary information to the SSS for performing search. After executing each query, the SSS sends searched records to the WSS for shuffling and re-randomising to provide privacy.

5.1 Solution Overview

In this chapter, we propose an SSE scheme for relational databases that can minimise the sensitive information leakage and resist against leakage-based attacks. In this section, we define the system and threat model and illustrate the techniques used in *P-McDb* in high-level.

5.1.1 System Model

In the following, we define our system model to describe the entities involved in *P-McDb*, as shown in Fig. 5.1:

- **Admin**: An admin is responsible for the setup and maintenance of databases, user management as well as specification and deployment of access control policies.
- **User**: It represents a user who joins the system. If granted by the admin, a user can issue insert, select, delete, and update queries according to deployed access control policies.
- **Storage and Search Service (SSS)**: It provides encrypted data storage, executes encrypted queries and returns matching records in an encrypted manner.
- **Witness and Shuffle Service (WSS)**: It stores the auxiliary information needed for retrieving data. After executing each query, it shuffles and re-randomises searched records and auxiliary information to achieve the privacy of access pattern. The WSS has no access to the encrypted data.

- **Cloud Service Providers (CSPs)**: Each of the WSS and the SSS is deployed on the infrastructure managed by a separate CSP. The CSPs have to ensure that there is a two-way communication between the WSS and SSS, but our model assumes there is no collusion between the CSPs.

5.1.2 Threat Model

We assume the admin is fully trusted. All the users are only assumed to securely store their keys and the data.

The CSPs hosting the SSS and the WSS are modelled as honest-but-curious. More specifically, they honestly perform the operations requested by users according to the designated protocol specification. Meanwhile, they are curious to learn sensitive information by analysing the stored and exchanged data, or injecting malicious data. However, we do not consider active attacks, such as modifying the database and exchanged data, and assume that there are mechanisms in place for ensuring data integrity and availability of the system. Moreover, we assume both the SSS and the WSS are part of the public cloud infrastructures provided by different CSPs.

As assumed in [42, 64, 97], we also assume the CSPs do not collude. In practice, cloud providers with conflict interests, such as Amazon S3, Google Drive and Microsoft Azure, could be considered since they may be less likely to collude in an attempt to gain information from their customers.

5.1.3 Approach Overview

P-McDb aims at hiding the search, access, and size patterns. *P-McDb* also achieves both backward and forward privacy. We now give an overview of our approach.

To protect the search pattern, in *P-McDb*, each query is XORed with a nonce, making the encrypted query semantically secure. That is, identical queries will look different once encrypted. However, the SSS may still infer the search pattern by looking at the access pattern. That is, by looking at the physical locations and ciphertext of the encrypted records returned by the search operation, the SSS can infer that two queries searching for the same fields are equivalent if the same records are returned. To address this issue, after executing each query, *P-McDb* also shuffles the locations of the searched records. Moreover, prior to re-inserting these records, *P-McDb* re-randomises the ciphertexts, making them untraceable. Technically, after each search operation, we update all the searched records in the database. In this way, even if a query equivalent to the previous one is executed, the SSS will see a new set of records being searched and returned, and cannot easily infer the search and access pattern.

Another form of leakage is the size pattern leakage, where an adversary can learn the number of records returned after performing a query. Even after the shuffling and re-randomising, an adversary could still count the number of records matching each query and guess whether two queries are equivalent by checking the number of records in the result set. To address this issue, as done in *ObliviousDB*, the elements in each field are divided into groups with a deterministic function and the elements in the same group are padded into the same occurrence with dummy records. Moreover, *P-McDb* also adapts *Solution 1* introduced in Section 3.3.2 to manage the groups when inserting, deleting or updating records. Here we clarify that the search pattern is not fully protected in *P-McDb*. Specifically, the SSS can still tell if the queries are in the same group or not based on the number of records matching them.

Furthermore, due to the nonces, *P-McDb* also achieves forward and backward privacy. After each search operation, the records are re-randomised using fresh nonces. Only queries that include the current nonce will be able to match the records. In this way, even if a malicious CSP tries to use previously executed queries with old nonces, she will not be able to match the records in the data set, ensuring forward privacy. Similarly, deleted records (with old nonces) will not match newly issued queries because they use different nonces.

At a high-level, *ObliviousDB* uses the same ideas as *P-McDb* to protect the search, access, and size patterns and to ensure forward and backward privacy. However, the design details are completely different. The details and algorithms of *P-McDb* will be discussed in the following section.

5.2 Solution details

5.2.1 Setup

The system is set up by the admin by generating the secret keys s_1 and s_2 based on the security parameter k. s_1 is only known to the user and is used to protect the data (including queries and records) from both the SSS and the WSS. s_2 is known to both the user and the WSS and is used to generate nonces for record and query encryption. The admin also defines a symmetric encryption scheme $Enc : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^*$, such as AES-ECB, which ensures the encrypted data is searchable and retrievable. This is sufficient for our searchable encryption functionality; we do not require any special features of the encryption scheme.

Group Generation. If the initial database is not empty, as done in *ObliviousDB*, the admin of *P-McDb* also divides the elements in each field into groups with a deterministic function, pads the elements in the same group into the same occurrence by generating dummy records, and builds the index information for each group. The difference is, in *P-McDb*, the elements in each group and the index information are sent to the WSS.

Table 5.1: Data representation in *P-McDb*.

(a) *Staff*

Name	**Age**
Alice	25
Anna	30
Bob	27
Bill	25
Bob	33
Baker	25

(b) GDB on the WSS

GID	**Index List**	**Elements**
$(1, g_{1,1})$	$\{1,2\}$	$E_{1,1} = \{Enc_s(Alice), Enc_s(Anna)\}$
$(1, g_{2,1})$	$\{3,4,5,6,7,8\}$	$E_{2,1} = \{Enc_s(Bob), Enc_s(Bill), Enc_s(Baker)\}$
$(2, g_{1,2})$	$\{1,3,4,6,7,8\}$	$E_{1,2} = \{Enc_s(25), Enc_s(27)\}$
$(2, g_{2,2})$	$\{2,5\}$	$E_{2,2} = \{Enc_s(30), Enc_s(33)\}$

(c) NDB on the WSS

id	***seed***
1	$seed_1$
2	$seed_2$
3	$seed_3$
4	$seed_4$
5	$seed_5$
6	$seed_6$
7	$seed_7$
8	$seed_8$

(d) EDB on the SSS

ID	**1**	**2**	**Tag**
1	$SE(Alice)$	$SE(25)$	tag_1
2	$SE(Anna)$	$SE(30)$	tag_2
3	$SE(Bob)$	$SE(27)$	tag_3
4	$SE(Bill)$	$SE(25)$	tag_4
5	$SE(Bob)$	$SE(33)$	tag_5
6	$SE(Baker)$	$SE(25)$	tag_6
7	$SE(Bill)$	$SE(27)$	tag_7
8	$SE(Baker)$	$SE(27)$	tag_8

(a) A sample *Staff* table. (b) GDB, the group information, is stored on the WSS. (c) NDB, containing the seeds used to generate nonces, is stored on the WSS. (d) EDB, the encrypted *Staff* table, is stored on the SSS. Each encrypted data element $SE(D_m) = Enc_{s_1}(e_f) \oplus n_f$. Each record has a tag, enabling users to distinguish between dummy and real records.

Algorithm 6 $RcdEnc(rcd, flag, s_1, s_2)$

1: $seed \xleftarrow{\$} \{0,1\}^{|seed|}$, where $|seed| < |e|$
2: $n \leftarrow \Gamma_{s_2}(seed)$, where $n = n_1||...||n_F||n_{F+1}$, $|n_f| = |e_f|$ and $|n_{F+1}| = 2k$
3: **for** each element $e_f \in rcd$ **do**
4: $\quad \sigma \leftarrow H_{s_1}(e_f)$
5: $\quad GE(e_f) \leftarrow PRF(\sigma)$
6: $\quad SE(e_f) \leftarrow Enc_{s_1}(e_f) \oplus n_f$
7: **if** $flag = 1$ **then**
8: $\quad S \xleftarrow{\$} \{0,1\}^k$, $tag \leftarrow (H_{s_1}(S)||S) \oplus n_{F+1}$
9: **else**
10: $\quad tag \xleftarrow{\$} \{0,1\}^{2k}$
11: **return** $Ercd = (SE(e_1),...,SE(e_F),tag)$, $seed$ and $Grcd = ((GE(e_1),Enc_{s_1}(e_1)),$ ..., $(GE(e_F),Enc_{s_1}(e_F)))$

Data Encryption. In this part, we describe the steps involved in encrypting a record in *P-McDb*. Basically, for each record, the admin generates the group information for each element and encrypts it. The details of each operation are provided in Algorithm 6, *RcdEnc*.

***RcdEnc*$(rcd, flag, s_1, s_2) \rightarrow (Grcd, Ercd)$.** *RcdEnc* takes the record $rcd = (e_1,...,e_F)$, the $flag$ marking if rcd is real or dummy ($flag = 1$ when rcd is real, otherwise $flag = 0$), and the keys s_1 and s_2 as the inputs, and outputs the encrypted record $Ercd$, a seed, and the group information $Grcd$. To encrypt a (real or dummy) record rcd, first the admin or the user generates a random string as a $seed$ for generating a longer nonce n with a pseudo-random generator (PRG) $\Gamma : \{0,1\}^k \times \{0,1\}^{|seed|} \rightarrow \{0,1\}^*$ (Line 2, Algorithm 6). The user does not have to remember the seed $seed$; this is the job of the WSS. For each data element e_f in rcd, the admin generates its group information by running $GE(e_f)$ (Line 5). Moreover, e_f is encrypted using the symmetric encryption algorithm Enc, and then XORed with a nonce fragment n_f (Line 6). The use of n_f ensures the forward and backward privacy of $SE(e_f)$ (The reason is given in Section 5.2.3). $SE(e_f)$ will be used for encrypted search and data retrieval.

Considering both the WSS and the SSS are untrusted in *P-McDb*, we cannot mark the real and dummy records with 1 and 0 in cleartext. Instead, we use a keyed hash value to mark if each record is real or dummy. Specifically, as shown in Lines 8 and 10, a tag tag is generated using a keyed hash function $H : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ and the secret key s_1 if the record is real, otherwise tag is a random bit string. With the secret key s_1, the dummy records can be efficiently filtered out by users before decrypting the search result by checking if:

$$lhtag \stackrel{?}{=} H_{s_1}(rhtag), \text{ where } lhtag||rhtag \leftarrow tag \oplus n_{F+1}$$

Once all the real and dummy records are encrypted, the admin uploads the group information and the seed for each record to the WSS, and each encrypted record to the SSS. The group

information stored in the WSS is called *GDB*, and the seed information stored in the WSS is called in *NDB*. The encrypted database stored in the SSS is called *EDB*. Table 5.1 provides an example of *GDB*, *NDB*, and *EDB* for the *Staff* table shown in cleartext in Table 5.1(a). In our example, the *Staff* table has 2 fields and contains 6 records.

GDB (shown in Table 5.1(b)) contains a list of indices and the encrypted elements for each group. *NDB* (shown in Table 5.1(c)) contains a seed for each record stored in *EDB*. *EDB* (shown in Table 5.1(d)) contains the encrypted records. Note that to ensure the correctness of the search functionality, it is necessary to store the encrypted records and their respective seeds in the same order in *EDB* and *NDB* (the search operation in details in given in Section 5.2.2). In addition to the database fields (*i.e., Name* and *Age*), *EDB* also contains the tag field for *tag*. Note that *EDB* contains 2 extra records with *id* 7 and 8: they are the dummy records generated by the admin.

5.2.2 Select Query

For simplicity, we also only explain our solution for queries whose predicate is a single equality predicate. As mentioned in Section 4.2, the complex queries can be performed by checking the predicate one by one, and whether a record matches the query can be determined by combining each check result according to the conjunctives and disjunctives between predicates. To support range queries, the technique used in [90] can be adopted, where numbers are converted into several bags of bit strings, and an inequality over a number is converted into several equalities over bit strings.

For performing a select query, *P-McDb* requires the cooperation between the WSS and the SSS. The details of the steps performed by the user, the WSS, and the SSS are shown in Algorithm 7.

***Query*$(Q, s_1, s_2) \rightarrow$ *SR*.** On the user side (Lines 1 - 9, Algorithm 7), the algorithm takes the query $Q = (type, f, operator, e)$ and the secret key s_1 as the inputs, and outputs the encrypted query EQ and a nonce η. To ensure the search efficiency, the user first generates the group identifier $GE(Q.e)$ by computing $PRF(Q.e))$ (Line 3), which is defined in Section 4.2. Recall that we do not aim at protecting the query operator, type and the field pattern from CSPs. Thus, the user does not encrypt $Q.operator$, $Q.type$, and $Q.f$. The interested keyword $Q.e$ is first encrypted with the deterministic encryption Enc using s_1, and then randomised with a nonce η (Line 7). The nonce ensures that the encrypted keyword $EQ.e^*$ is semantically secure. Finally, the user sends the encrypted query EQ to the SSS and sends $EQ.f$, η, and $GE(Q.e)$ to the WSS.

Given $EQ.f$, η, and $GE(Q.e)$, the WSS provides the list of indices of the searched group IL and the corresponding nonces EN to the SSS (Line 10 - 18). Specifically, the WSS first gets

Algorithm 7 Query(Q, s_1, s_2)

User(Q, s_1):
$\sigma \leftarrow H_{s_1}(Q.e)$
$GE(Q.e) \leftarrow PRF(\sigma)$
$EQ.type \leftarrow Q.type$
$EQ.operator \leftarrow Q.operator$
$EQ.f \leftarrow Q.f$
$\eta \xleftarrow{\$} \{0,1\}^{|e|}$, $EQ.e^* \leftarrow Enc_{s_1}(Q.e) \oplus \eta$
Send EQ to the SSS
Send $(EQ.f, \eta, GE(Q.e))$ to the WSS

$WSS(EQ.f, \eta, GE(Q.e), s_2)$:
$EN \leftarrow \emptyset$
$IL \leftarrow GDB(EQ.f, GE(Q.e))$
for each $id \in IL$ **do**
    $n \leftarrow \Gamma_{s_2}(seed_{id})$, where $n = ...||n_{EQ.f}||...$ and $|n_{EQ.f}| = |\eta|$
    $w \leftarrow H(n_{EQ.f} \oplus \eta)$
    $t \leftarrow \eta \oplus seed_{id}$
    $EN(id) \leftarrow (w,t)$
Send IL and the encrypted nonce set EN to the SSS

$SSS(EQ, EN, IL)$:
$SR \leftarrow \emptyset$
for each $id \in IL$ **do**
    if $H(EDB(id, EQ.f) \oplus EQ.e^*) = EN(id).w$ **then**
        $SR \leftarrow SR \cup (EDB(id), EN(id).t)$
Send the search result SR to the user

User(SR, η, s_2):
for each $(Ercd, t) \in SR$ **do**
    $n \leftarrow \Gamma_{s_2}(t \oplus \eta)$
    $(Enc_{s_1}(rcd), tag) \leftarrow Ercd \oplus n$
    $lhtag||rhtag \leftarrow tag$, where $|lhtag| = |rhtag|$
    if $lhtag = H_{s_1}(rhtag)$ **then**
        $rcd \leftarrow Enc_{s_1}^{-1}(Enc_{s_1}(rcd))$

the IL of group $GE(Q.e)$ from GDB (Line 12). Second, for each $seed_{id}$ indexed by IL, the WSS recovers the nonce n by computing $\Gamma_{s_2}(seed_{id})$ and creates a tuple $EN(id) = (w,t)$ (Line 17). In particular, $w = H(n_{EQ.f} \oplus \eta)$ will be used by the SSS to find the matching records, and $t = \eta \oplus seed_{id}$ will be used by the user to decrypt the result. For generating the witness w, since only the $seed$ is stored in NDB, the WSS has to replay the PRG function to recover the nonce for each record (Line 14), which increases the computation overhead on the WSS. To improve the performance of the WSS, the admin can also send the nonce n blinded in each record to the WSS directly. Whereas, storing the longer nonces puts more storage overhead on the WSS. In practice, whether caching the nonces on the WSS or not can be determined by the WSS storage capacity and the requirement for the system performance.

For each query, the SSS gets the encrypted query EQ from the user and gets IL and EN from the WSS. It traverses the records indexed by IL and finds the records matching EQ with the assistance of EN (Line 19 - 24). Specifically, the SSS first creates an empty set SR to contain search results. The SSS performs the search operation over EDB. For each record indexed by IL, the SSS checks if $H(EDB(id, EQ.f) \oplus EQ.e^*) \stackrel{?}{=} EN(id).w$ (Line 22). In details, the operation is:

$$H(Enc_{s_1}(e_{EQ.f}) \oplus n_{EQ.f} \oplus Enc_{s_1}(Q.e) \oplus \eta) \stackrel{?}{=} H(n_{EQ.f} \oplus \eta)$$

As long as the encrypted records and their respective seeds have the same identifiers, only when $Q.e = e_{EQ.f}$ there is a match. Then, the SSS adds the matched $(EDB(id), EN(id).t)$ to SR (Line 23). Finally, the SSS sends SR to the user.

Only the user issuing the query knows η and is able to recover the nonce n for each returned record by computing $\Gamma_{s_2}(t \oplus \eta)$ (Line 27). With n, the user can check if each returned record is real or dummy (Line 30), and decrypt each real record by computing $Enc_{s_1}^{-1}(EDB(id) \oplus n)$ (Line 31), where Enc^{-1} is the inverse of Enc.

Discussion. Based on the above description, the SSS needs a witness w to check if the record matches the query. Since each record in EDB is bound with a unique nonce, the WSS has to send all the blinded nonces indexed by IL to the SSS for each query. When IL is large, the communication overhead between the WSS and the SSS will be computationally intensive. To address this issue, the elements in the same groups can be blinded by the same nonces. In this way, the WSS just needs to generate one pair of (w,t) for each query.

Note that the nonce blinded in each record is mainly used to ensure the forward and backward privacy, rather than to hide the statistical information of the database. Indeed, the statistical information of the database is already protected by the dummy records. As long as the dummy records are indistinguishable from the real ones, the WSS and the SSS can only learn the elements in the same group have the same occurrences. The only issue is the SSS and

the WSS can learn if the records share the same elements or not, since the Enc encryption is deterministic. The fact is that, even if the encrypted records are semantically secure, the SSS can still learn the matched records have the same element in the searched fields after executing equality queries. Therefore, blinding the records in the same group with the same nonce does not leak any additional information to the WSS and the SSS.

5.2.3 Shuffling and Re-randomisation

Algorithm 8 $Shuffle(IL, s_2)$

1: $Ercds \leftarrow EDB(IL)$ {Get from the SSS all the records indexed by IL}
2: **for** each $Ercd_{id} \in Ercds$ **do**
3: $n \leftarrow \Gamma_{s_2}(seed_{id})$
4: $Ercd_{id} \leftarrow Ercd_{id} \oplus n$
5: $seed \xleftarrow{\$} \{0,1\}^{|seed|}$
6: $r \leftarrow \Gamma_{s_2}(seed)$, where $|r| = M|D| + 2\lambda$
7: $Ercd_{id} \leftarrow Ercd_{id} \oplus r$
8: $NDB(id) \leftarrow seed$
9: Shuffle $Ercds$ and update NDB
10: Update the indices of affected groups
11: Send $Ercds$ to the SSS.

To protect the access pattern and ensure the forward and backward privacy, our solution is shuffling and re-randomising the searched records after executing each query, as done in *ObliviousDB*. In *P-McDb*, the WSS is responsible for shuffling and re-randomising the searched records. The details are shown in Algorithm 8, *Shuffle*.

***shuffle*$(IL, s_2) \rightarrow Ercds$.** The WSS first gets all the searched records from the SSS. To prevent the SSS learning if any two queries in the same group match the same records or not, the first step is to update the ciphertext of all the searched records by re-randomising them (Lines 2 - 8). Specifically, for each record in $Ercds$, the WSS first recovers the respective nonce (Line 3)[1] and removes the nonce included in each element (line 4). Second, the WSS samples a new seed, generates a new nonce, and randomises the record with the new nonce (Line 7). Meanwhile, the respective seed stored in NDB is updated to the new seed (Line 8).

The second step to protect the access pattern is to change the storage locations of the searched records by shuffling them (Line 9). In our implementation, we leverage the modern version of the Fisher-Yates shuffle algorithm [98], where from the first record to the last one, the storage location of each record is exchanged with a random record storing behind it. Note that the shuffling operation affects the list of indices of the groups in other fields. After shuffling the records, the WSS also updates the index lists of other groups accordingly (Line 10). Finally, the re-randomised and shuffled records $Ercds$ are sent back to the SSS.

[1] This step can be skipped if the WSS caches the nonces.

By using a new set of seeds during the re-randomising, we are able to achieve both forward and backward privacy. If the SSS tries to execute an old query, it will not be able to match any records without the new seeds, which is known only to the WSS. Similarly, the SSS cannot learn if deleted records match new queries.

5.2.4 User Revocation

Because of the nonces bound to SE, without the assistance of the WSS and the SSS, the revoked user is unable to recover the query and records only with s_1. Therefore, for user revocation, we just need to manage a revoked user list at the WSS as well as at the SSS. Once a user is revoked, the admin informs the WSS and the SSS to add this user into their revoked user lists. When receiving a query, the WSS and the SSS will first check if the user has been revoked. If yes, they will reject the query. In case a revoked user colludes with either the SSS or WSS, she cannot get the search results, since such operation requires the cooperation of both the user issuing the query, the WSS, and the SSS.

5.2.5 Database Updating

P-McDb also uses *Solution 1* given in Section 3.3.1 to manage the dummy records. Basically, when inserting or deleting records, the user ensures the elements in the same groups still have the same occurrences by inserting, deleting or updating dummy records. The details for inserting a record is shown in Algorithm 9.

$\boldsymbol{Insert(rcd, s_1, s_2) \rightarrow (Insert_{WSS}, Insert_{SSS}, Update)}$**.** In *P-McDb*, the insert query is also performed with the cooperation of the user, the WSS, and the SSS. Specifically, the user first encrypts the record $rcd = (e_1, ..., e_F)$ by running *RcdEnc*. Note here if the elements in the same group are encrypted with the same nonce, the user should send the *Grcd* to the WSS and get the respective seeds or nonces for encryption then.

Second, to minimise the number of required dummy records, for each field f, the user selects all the records containing e_f and checks if one of them is dummy (Lines 6 - 10).

If yes, for each $e_f \in rcd$, the user updates one of the dummy records containing e_f to $SE(NULL)$ (Lines 11 - 14). As a result, the occurrence of e_f is unchanged. Since we do not consider the forward and backward privacy of 'NULL', the user uses the same nonce in $SE(e_f)$ for generating $SE(NULL)$ (the nonce can be recovered from the returned t_{id}). By doing so, both the user and the WSS do not need to update the seed. All the records to be updated will be sent to the SSS.

On the contrary, *i.e.,* at least one of the inserted element is not included in a dummy record, the user will increase its group threshold by one and increase the occurrence of all the other elements in this group by one by inserting dummy records (Lines 15 - 25). Specifically, the user

Algorithm 9 $Insert(rcd)$

$\underline{user_i(rcd, s_1, s_2)}$:
$Update \leftarrow \emptyset$, $Insert_{WSS} \leftarrow \emptyset$, $Insert_{SSS} \leftarrow \emptyset$
$(Ercd, seed, Grcd) \leftarrow RcdEnc(rcd, 1, s_1, s_2)$
$Insert_{WSS} \leftarrow (seed, Grcd)$
$Insert_{SSS} \leftarrow Ercd$
$sign \leftarrow 0$
for each $e_f \in rcd$ **do**
    $SR_f \leftarrow Query(e_f)$
    if All the records in SR_f are real **then**
        $sign = 1$, break;
if $sign = 0$ **then**
    for $f = 1$ to F **do**
        Assume $Ercd_{id} \in SR_f$ is dummy, $Ercd_{id}.e_f^* \leftarrow SE(NULL)$
        $Update \leftarrow Update \cup Ercd_{id}$
else
    for each $(f, GE(e_f)) \in Grcd$ **do**
        $E_{f,GE(e_f)} \leftarrow GDB(f, GE(e_f))$
    $M = \max\{|E_{f,GE(e_f)}| \mid 1 \leq f \leq F\}$
    for $m = 1$ to M **do**
        Generates $rcd_m = (NULL, ..., NULL)$
    Assign the elements in $(E_{1,GE(e_1)}, ..., E_{F,GE(e_F)})$ to $\{rcd_1, ..., rcd_M\}$
    for $m = 1$ to M **do**
        $(Ercd, seed, Grcd) \leftarrow RcdEnc(rcd_m, 0, s_1, s_2)$
        $Insert_{WSS} \leftarrow Insert_{WSS} \cup (seed, Grcd)$
        $Insert_{SSS} \leftarrow Insert_{SSS} \cup Ercd$
Send $Insert_{WSS}$ to the WSS
Send $Insert_{SSS}$ and $Update$ to the SSS

$\underline{SSS(Insert_{SSS}, Update, EDB)}$:
$IDs \leftarrow \emptyset$
for each $Ercd \in Insert$ **do**
    $EDB(++id) \leftarrow Ercd$
    $IDs \leftarrow IDs \cup id$
for each $Ercd \in Update$ **do**
    Update $EDB(Ercd.id)$ with $Ercd$
Send IDs to the WSS

$\underline{WSS(Insert_{WSS}, GDB, IDs)}$:
for each $(seed, Grcd) \in Insert$ and $id \in IDs$ **do**
    $NDB(id) \leftarrow seed$
    for $f = 1$ to F **do**
        $GDB(GE(e_f)) \leftarrow GDB(GE(e_f)) \cup id$

first gets the elements $E_{f,GE(e_f)}$ for each involved group $(f, GE(e_f))$ from the WSS. As shown in Table 5.1(b), $E_{f,GE(e_f)}$ contains all the encrypted elements in the group. Assume the biggest one of the F involved groups contains M elements, *i.e.*, $M = \{|E_{f,GE(e_f)}| \mid 1 \leq f \leq F\}$. Second, the user generates M dummy records that cover all the elements in $(E_{1,GE(e_1)}, ..., E_{F,GE(e_F)})$. In other words, the user uses the elements in $(E_{1,GE(e_1)}, ..., E_{F,GE(e_F)})$ to assemble the M dummy records. Third, the user encrypts all the dummy records with *RcdEnc* and gets $Ercd$, *seed*, and $Grcd$.

All the encrypted records, including both the real and dummy records, are held in $Insert_{SSS}$ and sent to the SSS. All the seeds and group information are held in $Insert_{WSS}$ and sent to the WSS. Note that $Ercd$ and $(seed, Grcd)$ for the same record are stored in the same order in $Insert_{SSS}$ and $Insert_{WSS}$.

The SSS inserts and updates the records included in $Insert_{SSS}$ and $Update$ (Lines 28 - 35), For each record in $Insert_{SSS}$, the SSS assigns an identifier *id* and stores it to $EDB(id)$ (Line 31). Recall that, to ensure the correctness of the search operation, the records in *EDB* and their respective seeds in *NDB* should have the same identifiers. Moreover, the identifier of $Ercd$ should be added into the indices in *GDB*. Thus, the SSS sends the identifier for each $Ercd$ to the WSS, so that the WSS can know the identifier and storage location for each *seed* in *NDB*. If $Update$ is not empty, the SSS updates each record with the new encrypted value (Line 34).

After getting the identifiers from the SSS, the WSS starts to process the *seed* and *Grcd* in $Insert_{WSS}$ (Lines 36 - 40). For each *seed* in $Insert_{WSS}$, the WSS inserts it to *NDB* based on the respective identifier *id* get from the SSS (Line 37). For each $Grcd = (GE(e_1), ..., GE(e_F))$, the WSS adds *id* to each involved group $(f, GE(e_f))$ (Line 40).

Finally, to protect the access pattern, the shuffling and re-randomising operations over the involved groups will be performed between the WSS and SSS.

For delete queries, each matched record can be turned into dummy by replacing its *tag* with a random $2k$-bit string.

5.3 Security Analysis

In this section, we prove *P-McDb* protects the search, size, and access patterns from both the SSS and the WSS, and achieves the forward and backward privacy.

To ensure the performance of the system, *P-McDb* also suffers from the $\mathcal{L}0$ leakage. That is, both the SSS and WSS can learn the query type, operator and the field pattern. Moreover, they learn which group is searched for each query, and which records are in the same group. *P-McDb* only ensures the privacy of the search, access, and size patterns within groups. In other words, we show that the SSS cannot distinguish two queries that have the same structure in the sense of Definition 7.

5.3.1 The Security Analysis against the SSS

In *P-McDb*, the SSS plays the same role as the public CSP used in *ObliviousDB*, *i.e.,* stores the encrypted database and executes the queries. Therefore, as done in Section 4.3, to prove the search, access, and size patterns and the forward and backward privacy are protected from the SSS, we can use the same way to give the definition and proof.

In short, as done in Definition 9, we build a game between a PPT adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, where $\mathcal{A}$ plays as the SSS and performs the encrypted search as defined in *P-McDb*, and $\mathcal{C}$ plays as other entities and executes their functions as required. In the game, $\mathcal{A}$ chooses the database and designs two queries Q_1 and Q_2 in the same structure for the challenge. $\mathcal{C}$ bootstraps the database as done by the admin in *P-McDb*, and then encrypts one of the queries as done by the user. After that, $\mathcal{C}$ and $\mathcal{A}$ engage in the full protocol to run the encrypted query. If $\mathcal{A}$ follows the designated protocols honestly and does not modify any data, it will get the encrypted query EQ, the indices list of the interested group IL, a set of blinded nonces EN for searching, the search result SR, and the shuffled and re-randomised records $Ercds$. Based on this information, $\mathcal{A}$ guesses which query is picked by $\mathcal{C}$. Moreover, before and after the challenge phase, $\mathcal{A}$ could also issue queries and get the corresponding EQ, IL, EN, SR, and $Ercds$. If $\mathcal{A}$ cannot guess which query is encrypted by $\mathcal{C}$ in the challenge phase with non-negligible advantage, we say *P-McDb* achieves the privacy of the search, size and access pattern, and forward and backward privacy against the honest-but-curious SSS.

Theorem 2. If the WSS generates IL and EN and performs the shuffling correctly, *P-McDb* achieves the privacy of the search, size and access pattern, and forward and backward privacy against the SSS.

Proof. (Sketch) **Static database.** First, we assume the database is static and focus on the select queries. As discussed in Section 4.3, $\mathcal{A}$ cannot win the game based on EQ and IL, since Q_1 and Q_2 have the same structure and $EQ.e^*$ is randomised with a nonce.

Due to the seeds, all the encrypted records in SR and $Ercds$ are semantically secure. Moreover, since Q_1 and Q_2 are in the same group, both the $|SR|$ and $|Ercds|$ are fixed no matter which query is selected. In addition, if all the records in the searched group are shuffled randomly by $\mathcal{C}$ after executing each query, the identifiers of the records in SR will be independent of the selected query. Thus, $\mathcal{A}$ cannot win the game based on SR and $Ercds$.

Comparing with *ObliviousDB*, the SSS also gets EN in *P-McDb*. As described in Algorithm 7, the elements in EN are blinded with a nonce which is independent of the query. Thus, the adversary cannot win the game based on EN.

Dynamic database. If the database is dynamic, *P-McDb* and *ObliviousDB* use the same method to manage the dummy records when inserting and deleting records. For insert query,

the SSS in *P-McDb* and the CSP in *ObliviousDB* receive the same messages from the user, *i.e.,* the encrypted real record to be inserted and the dummy records to be updated or the new dummy records to be inserted. The records are encrypted in different ways in *P-McDb* and *ObliviousDB*, yet in both schemes, the encrypted records are semantically secure. Therefore, the security analysis for insert queries can be referred to the discussion in Section 4.3.

$\mathcal{A}$ could also send delete queries in phase 1, the challenge phase and/or phase 2. For delete queries, the $\mathcal{A}$ just need to assign a random $2k$-bit string to the tag of each matched record. By doing so, the real records will be turned into dummy. After shuffling and re-randomising, $\mathcal{A}$ cannot determine which record's tag has been revised. Therefore, $\mathcal{A}$ cannot learn additional information and win the game by issuing delete queries in phase 1, the challenge phase and/or phase 2. □

5.3.2 The Security Analysis against the WSS

In *P-McDb*, the WSS is also honest-but-curious. In this part, we prove the patterns and the forward and backward privacy are also protected from the WSS. In the security definition given below, we model the WSS as the honest-but-curious PPT adversary $\mathcal{A}$ who follows the designated protocols and gets the *GDB* and *NDB* stored on the WSS and the data the WSS received when executing a query, but does not mount active attacks.

Definition 10. Let $\prod_{P\text{-}McDb}$ =(Setup, Query, Shuffle) be *P-McDb*, and λ and k be the security parameters. $\mathcal{A}$ is a PPT adversary, and $\mathcal{C}$ is a challenger. The game between $\mathcal{A}$ and $\mathcal{C}$ in $\prod_{P\text{-}McDb}$ is described as below:

- **Setup** The challenger $\mathcal{C}$ first initialises the system by generating s_1 and s_2. The adversary $\mathcal{A}$ is only given s_2.

- **Bootstrap** $\mathcal{A}$ submits a database Δ. As done by the Admin, $\mathcal{C}$ encrypts Δ and divides the data in each field into groups. Moreover, $\mathcal{C}$ generates a number of dummy records for each group, such that the elements in each group have the same occurrences. The encrypted database EDB is securely kept by $\mathcal{C}$. The encrypted group information *GDB* and seeds *NDB* are sent to $\mathcal{A}$.

- **Phase 1** $\mathcal{A}$ can make polynomially many queries Q in plaintext. As would be done by the user, $\mathcal{C}$ generates the group information, encrypts and blinds each query Q with a nonce η, and sends the field $EQ.f$, the group $GE(Q.e)$ and η to $\mathcal{C}$. After that, $\mathcal{A}$ gets *IL* from *GDB* and generates *EN*, and sends them to $\mathcal{C}$. After the search operation, $\mathcal{C}$ and $\mathcal{A}$ engage in the shuffling and re-randomising to update *EDB*. So, for each query, $\mathcal{A}$ sees $GE(Q.e)$, $EQ.f$, η and all the searched records $Ercds$.

- **Challenge** $\mathcal{A}$ sends two queries Q_0 and Q_1 to $\mathcal{C}$ that have the same structure, which can be those already issued in phase 1. $\mathcal{C}$ and $\mathcal{A}$ perform the full protocol, so that $\mathcal{A}$ learns $GE(Q.e)$, $EQ.f$, η and $Ercds$.
- **Phase 2** $\mathcal{A}$ continues to adaptively request polynomially many queries, which could include the challenged queries Q_0 and Q_1.
- **Guess** $\mathcal{A}$ submits her guess b'.

The advantage of $\mathcal{A}$ in this game is defined as:

$$Adv_{\mathcal{A},\prod_{P\text{-}McDb}}(1^k) = Pr[b' = b] - \frac{1}{2}.$$

We say *P-McDb* achieves the privacy of the search, size and access pattern, and forward and backward privacy against the honest-but-curious WSS, if all PPT adversaries have a negligible advantage in the above game.

Theorem 3. If the WSS generates IL and EN and performs the shuffling correctly, *P-McDb* achieves the privacy of the search, size and access pattern, and forward and backward privacy against the WSS.

Proof. (Sketch) We show that the bit b chosen by $\mathcal{C}$ is hidden from the view of $\mathcal{A}$.

Static database. First, we consider the static database and focus on the select queries.

Consider the view of $\mathcal{A}$ in the game. $\mathcal{A}$ chooses an arbitrary database Δ and uploads this to $\mathcal{C}$. In Phase 1, $\mathcal{A}$ makes queries that are answered correctly by $\mathcal{C}$ by following the protocols.

In the challenge round, $\mathcal{A}$ sends two queries Q_0 and Q_1, and receives $EQ.f$, $GE(Q.e)$ and η. By definition, the two queries have the same structure. Moreover, the group generation is deterministic. Hence, the same $EQ.f$ and $GE(Q.e)$ will be received by $\mathcal{A}$ for either query. The nonce η is independent of the interested term in the query, so $\mathcal{A}$ cannot distinguish the query terms given η.

Finally, $\mathcal{A}$ gets all the searched records $Ercds$ from $\mathcal{A}$ for re-randomising and shuffling. For all the queries in the same group, if $\mathcal{A}$ follows the designated protocols honestly, it always gets the same set of records. Thus, $Ercds$ is also independent of b. For re-randomising records, $\mathcal{A}$ first removes the old nonces blinded in each searched record. After removing the nonces, the records are protected only with the deterministic encryption Enc. Thus, the WSS can learn if the records contain the same elements or not. Even though, due to the inserted dummy records, the real occurrence of each element is still hidden from the WSS.

The game continues in Phase 2. $\mathcal{A}$ may repeat Q_0 and/or Q_1. However, $\mathcal{A}$ always gets the same $GE(Q.e)$, $EQ.f$ and $Ercds$ for any query in the same group, and the nonce η is independent of the interested term included in the query.

Since $\mathcal{A}$ has no information to distinguish the bit b, the scheme satisfies the definition.

Dynamic database. Second, we consider the insert and delete queries in dynamic databases.

In phase 1, no matter what types of queries are issued by $\mathcal{A}$, as long as Q_0 and Q_1 are in the same structure, as discussed above, $\mathcal{A}$ cannot infer b based on $GE(Q.e)$, $EQ.f$ and η.

In the challenge phase, if Q_0 and Q_1 are two insert queries, $\mathcal{A}$ runs Algorithm 9. Basically, $\mathcal{C}$ first inserts Q_b ($Q_b = (e_1, ..., e_F)$ is a record). Second, for each element $e_f \in Q_b$, $\mathcal{C}$ selects all the records containing it and checks if one of them is dummy. If yes to all the F elements, for each e_f, $\mathcal{C}$ updates one dummy record containing e_f into 'NULL' in field f. Otherwise, $\mathcal{C}$ inserts dummy records to increase the occurrence of other elements each involved group. In both cases, $\mathcal{A}$ receives the group information *Grcd* of Q_b and a seed *seed* used to encrypt Q_b. Moreover, as done for select queries, $\mathcal{C}$ also gets a nonce η when searching each e_f. In the former case, no message is sent to $\mathcal{C}$ when updating the dummy records. In the latter case, $\mathcal{C}$ also gets *Grcd* and *seed* when inserting each dummy record. Indeed, as mentioned in Algorithm 9, the dummy records have the same elements as the real one. Hence, their *Grcd*s are equal to the real one. Recall that, for the records in the same structure, their elements in each field are in the same group, meaning *Grcd* is the same for either Q_0 or Q_1. Moreover, *seed* and η are independent of b, since they are random bit strings. Therefore, $\mathcal{A}$ cannot win the game when Q_0 and Q_1 are two insert queries.

If Q_0 and Q_1 are two delete queries, $\mathcal{C}$ first selects the records matching Q_b, and then updates them into dummy by updating their *tag*s in *EDB* with random $2k$-bit strings. $\mathcal{A}$ is not engaged in this operation. Only for the searching operation, $\mathcal{A}$ gets $GE(Q.e)$, $EQ.f$ and η. However, as mentioned above, $\mathcal{A}$ cannot distinguish the bit b based on them. □

5.4 Performance Analysis

5.4.1 Complexity Analysis of *P-McDb*

Although *P-McDb* only supports linear search, the search operation over each record is significantly efficient. In this part, we theoretically and empirically analyse the costs associated with the search, shuffle, and decryption operations over each record in *P-McDb*.

Let $\mathbf{XOR}_k$, $\mathbf{Enc}_k$, $\mathbf{Hash}_k$, $\mathbf{PRG}_k$, and $\mathbf{Enc}_k^{-1}$ represent the computation overhead of performing these operations with k bits key, respectively. Recall that F is the number of fields.

To execute a query Q, the user generates $GE(Q.e)$, encrypts the interested term $Q.e$ with *Enc* and XORs it with a nonce, and encrypts $Q.f$ with a hash function. Hence, the computation complexity of encrypting a predicate on the user side is $\mathbf{Enc}_k + \mathbf{XOR}_k + 2\mathbf{Hash}_k$.

Afterwards, for each record, the WSS recovers the nonce with the seed and generates (w,t) with XORs and hash operations. The computation complexity of this operation is $\mathbf{PRG}_k +$

Table 5.2: The computation and communication overhead for each record or each query.

Method	Operation	Computation cost per query	Communication cost per record
P-McDb	Search	User: $\mathbf{XOR}_k + \mathbf{Enc}_k + 2\mathbf{Hash}_k$ WSS: $\mathbf{PRG}_k + \mathbf{XOR}_k + \mathbf{Hash}_k + \mathbf{XOR}_{\lvert seed\rvert}$ SSS: $\mathbf{XOR}_k + \mathbf{Hash}_k$	$\lvert Hash\rvert + \lvert seed\rvert$ bits
	Decryption	User: $\mathbf{XOR}_{\lvert seed\rvert} + \mathbf{PRG}_k + F(\mathbf{XOR}_k + \mathbf{Enc}_k^{-1})$	$\lvert Ercd\rvert + \lvert seed\rvert$ bits
	Shuffle	WSS: $\mathbf{PRG}_k + (F+2)\mathbf{XOR}_k$	$2\lvert Ercd\rvert$ bits
PPQED	Search	User: $\mathbf{SUB}_N$ CSP1: $\mathbf{HA}_{N^2} + k\mathbf{HE}_k + \mathbf{SC}$ CSP2: $\mathbf{HE}_k + \mathbf{HD}_k$	$2k + \lvert Ercd\rvert$ bits
	Decryption	User: $F(\mathbf{HD}_k + \mathbf{SUB}_N)$	$2\lvert Ercd\rvert$ bits
	Oblivious data retrieval	CSP1: $F(\mathbf{HA}_{N^2} + \mathbf{HE}_k)$ CSP2: $F\mathbf{HD}_k$	$3\lvert Ercd\rvert$ bits
SisoSPIR	Search	User: $\mathbf{XOR}_T + \mathbf{XOR}_{\lvert rcd\rvert}$ CSP1: $T\mathbf{XOR}_{\lvert rcd\rvert}$ CSP2: $(T+1)\mathbf{XOR}_{\lvert rcd\rvert}$	$2\lvert rcd\rvert$ bits+ T integers

$\mathbf{XOR}_k$, $\mathbf{Enc}_k$, $\mathbf{Hash}_k$, $\mathbf{PRG}_k$, and $\mathbf{Enc}_k^{-1}$ represent the computation overhead to perform these operations with k bits key, respectively. $\mathbf{HE}_k$ and $\mathbf{HD}_k$ represent the computation overhead of performing homomorphic encryption and decryption with k bits keys, respectively. $\mathbf{SUB}_N$ is the computation overhead of the modular subtract operation over N. $\mathbf{HA}_{N^2}$ stands for the computation overhead of homomorphic addition in $\mathbb{Z}_{N^2}^*$. **SC** represents the Secure Comparison (SC) protocol proposed in [99]. $\mathbf{XOR}_T$ is the computation overhead of XORing T integers.

$\mathbf{XOR}_k + \mathbf{Hash}_k + \mathbf{XOR}_{|seed|}$.

Finally, the SSS checks if each record matches the predicate by executing one XOR and one hash operations, the computation of which is $\mathbf{XOR}_k + \mathbf{Hash}_k$. In this protocol, the user sends $|SE(e)|$-bit data to both the WSS and the SSS, and the WSS sends $|Hash| + |seed|$ bits to the SSS for each record in the interested group.

After searching, the SSS sends the results to the user. Each encrypted record is $|Ercd| = F|e| + 2k$ bits long. In addition, the $|seed|$-bit seed for each matched records should also be sent to the user for recovering the nonce.

Meanwhile, the SSS also sends the searched records to the WSS for shuffling. The WSS re-randomises each record with a nonce, but before that the WSS has to generate the nonce with a new seed, indicating the computation overhead on the WSS is $\mathbf{PRG}_k + (F+2)\mathbf{XOR}_k$. In this protocol, the communication between the WSS and the SSS is $|Ercd|$ bits per record.

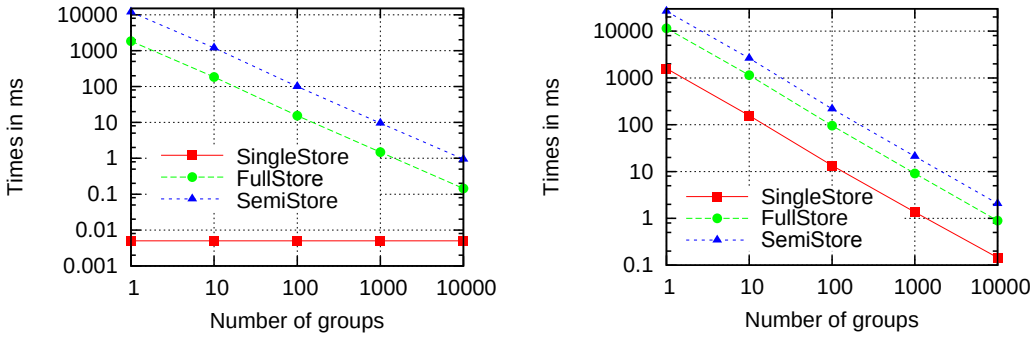
As mentioned in Section 2.2, only the HE-based $PPQED_a$ proposed by Samanthula *et al.* [42] and the ORAM-based SisoSPIR given by Ishai *et al.* [41] could hide the access pattern in databases. According to the description in [42] and [41], we also summarise the computation and communication overhead in $PPQED_a$ and SisoSPIR. From Table 5.2, we can infer that both the computation and communication overhead in our scheme is better than them. Note that $PPQED_a$ is based on public-key encryption system, and the lengths of its key k and $Ercd$

are much larger than the values in *P-McDb*.

5.4.2 Benchmark of *P-McDb*

We implemented *P-McDb* in C using MIRACL 7.0 library for cryptographic primitives. The performance of all the entities was evaluated on a desktop machine with Intel i5-4590 3.3 GHz 4-core processor and 8GB of RAM. As done in *ObliviousDB*, we also evaluated the performance using TPC-H benchmark [86], and tested equality queries with one singe predicates over 'O_CUSTKEY' field in 'ORDERS' table. Moreover, we divided the elements in 'O_CUSTKEY' field into 1, 10, 100, 1000, and 10000 groups by setting b =1, 10, 100, 1000, and 10000, respectively. When dividing the records into 1, 10, 100, 1000, and 10000 groups, the searched groups contains 4099836, 410000, 34000, 3300, and 320 records, and their thresholds are 41, 41, 34, 33, and 32, respectively. The specific details of the groups can be found in Section 4.4. In the following, all the results are averaged over 100 trials.

5.4.2.1 The Setting on WSS



(a) The *NonceBlind* running time on the WSS.

(b) The *Shuffle* running time on the WSS.

Fig. 5.2. The performance of the WSS in different cases.
In 'FullStore' case, the WSS stores both the seed and the nonce for each record. In 'SemiStore' case, the WSS only stores the seeds. In 'SingleStore' case, the elements in each group are blinded with the same nonce and the WSS only stores one seed for each group.

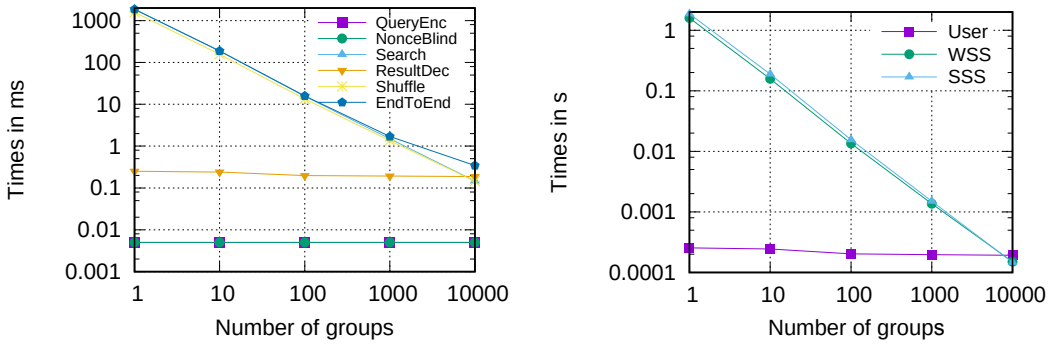
As mentioned in Section 5.2, the performance of the WSS depends on whether the nonces are stored in NDB and whether the elements in the same group are blinded with the same nonces. Thus, we first compared the performance of the WSS in three different cases. Specifically, in the first case, the WSS stores both the seed and the nonce for each record. We call this case 'FullStore'. In the second case, the WSS only stores the seeds and it has to derive the nonces for each query. We name this case 'SemiStore'. We call the last case as 'SingleStore', where the elements in the same group are blinded by the same nonce and the WSS only stores

one seed for each group. In *P-McDb*, the WSS is mainly responsible for executing the *Nonce-Blind* and the *Shuffle* operations for each query. Hence, we measured the running times of the two functions in the three cases and show the results in Fig. 5.2(a) and 5.2(b), respectively.

From Fig. 5.2(a) and 5.2(b), we can see that 'SingleStore' has the best performance for both the *NonceBlind* and the *Shuffle* operations. Moreover, the *NonceBlind* execution time is not affected by the groups in 'SingleStore' case, which is 0.005 milliseconds (ms). That is because the elements in the same group are blinded with the same nonces, the WSS just needs to generate one pair of (w,t) for each query. However, in the other two cases, the WSS has to generate the (w,t) pairs for all the records in the searched group, and the number of records in each group decreases when increasing the number of groups. Hence, the *NonceBlind* execution times of the two cases go down with the group number. In addition, Fig. 5.2(a) and 5.2(b) show that 'FullStore' performs better than 'SemiStore'. The reason is that in 'SemiStore' the WSS has to generate the nonce from the seed for each record, whereas, in 'FullStore' the WSS can get it from NDB directly.

In the following experiments, we use 'SingleStore' case.

5.4.2.2 The Performance of Each Operation



(a) The performance of each operation with different group numbers.

(b) The performance of each entity with different group numbers.

Fig. 5.3. The performance of each operation and entity.

To show the performance of *P-McDb* in details, we measured the execution time of each operation involved in the query process, including the query encryption, nonce blinding, searching, result decrypting, and the shuffling.

In Fig. 5.3(a), we show the effect on each operation when we change the number of groups. In this test, the tested query matches 34 records (32 real and 2 dummy records), which is also the occurrence of the elements in the searched group. As we can see from Fig. 5.3(a), the times for encrypting the query and blinding the nonce do not change with the number of groups. On

the contrary, both the searching and shuffling times drop sharply with the increase of the group number. The reason is that the search and the shuffle operations are performed over all the records in the searched group. The more groups, the fewer records in each group. Thanks to the efficient XOR operation, even when $g = 1$, *i.e.,* searching and shuffling the whole database (contains 4099836 records in total), the search and the shuffle operations can be finished in 1.87 and 1.59 seconds, respectively.

Moreover, we can also notice that the time taken by the result decryption decreases slowly when increasing the number of groups. For recovering the required records, in *P-McDb*, the user first filters out the dummy records and then decrypts the real records. Therefore, the result decryption time is affected by the number of returned real records as well as the dummy ones. In this experiment, the tested query always matches 32 real records. However, when changing the number of groups, the number of returned dummy records will be changed. Recall that, to break the link between the search and size patterns, all the elements in the same group are padded into the same occurrence, *i.e.,* the group threshold, with dummy records. The group threshold is determined by the highest occurrence in the group. When the records are divided into more groups, fewer elements will be included in each group. As a result, the occurrence of the searched element tends to be closer to the group threshold, and then fewer dummy records are required for its padding. Thus, the result decryption time decreases with the increase of the group number. In the tested dataset, the elements have very close occurrences, which ranges from 1 to 41. The number of matched dummy records are 9, 9, 2, 1 and 0 when there are 1, 10, 100, 1000 and 10000 groups, respectively. For the dataset with a bigger element occurrence gap, the result decryption time will change more obviously when changing the number of groups.

In addition, Fig. 5.3(a) also shows the end-to-end latency on the user side when issuing a query. In this test, we did not simulate the network latency, so the end-to-end latency shown here consists of the query encryption time, the nonce blinding time, the searching time and the result decrypting time. As shown in the figure, the query encryption and nonce blinding times are only 0.005 ms, which are negligible compared with the searching time. Thus, the end-to-end latency is dominated by the searching time and also decreases when increasing the number of groups. Note that the shuffle operation does not affect the end-to-end latency on the user side since it is performed by the WSS after the search operation.

5.4.2.3 The Performance of Each Entity

An important aspect of an outsourced service is that most of the intensive computations should be off-loaded to the CSPs. To quantify the workload on each of the entities, we measured the latency on the user, WSS, and SSS for processing the query with different numbers of groups. The results are shown in Fig. 5.3(b). We can notice that the computation times taken

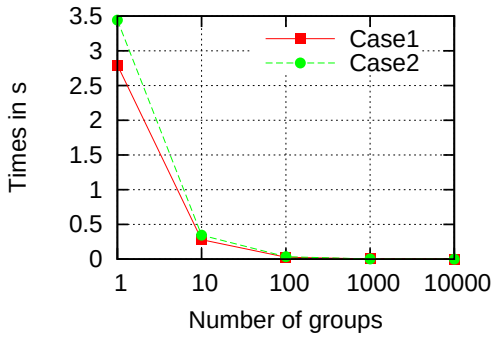


Fig. 5.4. The performance of the insert query.

on the WSS and the SSS are much higher than that on the user side. Moreover, in this test, the elements in the same groups are blinded by the same nonces. Thus, the shuffling time and the nonce blinding time taken on the WSS are improved significantly, making the computation overhead on the WSS a bit less than that on the SSS. The latency on the user side is mainly for decrypting the returned records, which will increase with the number of returned records. Based on our experiment, when the user gets more than 100000 records and setting $g = 10$, the user has the similar computation overhead as the WSS and the SSS.

5.4.2.4 Comparison with Other Schemes

To better investigate the performance of our approach, here we roughly compare the search time of *P-McDb* with $PPQED_a$ and SisoSPIR. Although we did not access their implementation, our experiments were conducted on Linux machines with approximate power[1]. Searching over 1 million records takes more than 10 seconds in SisoSPIR. In $PPQED_a$, it takes 1 second to check if a predicate matches with a record when the data size is 10 bits. However, *P-McDb* only takes less than 2 seconds when searching over 4.1 million records, which is much more efficient than the other two schemes.

5.4.2.5 The Performance of Insert Query

Since *P-McDb* is a dynamic SE scheme, we also tested its performance for insert queries. As done in *ObliviousDB*, *P-McDb* also utilises *Solution 1* to manage the dummy records when inserting and deleting records. Recall that, *Solution 1* processes the insert query in two different ways: 'Case1' and 'Case2'. In 'Case1', the user just needs to update F dummy records. However, in 'Case2', the user has to insert M dummy records. We also tested the performance of the two cases for *P-McDb* and show the result in Fig. 5.4.

[1] $PPQED_a$ was tested on a Linux machine with an Intel Xeon 6-Core CPU 3.07 GHz processor and 12 GB RAM and SisoSPIR was tested on a machine with an Intel i7-2600K 3.4GHz 4-core CPU 8GB RAM.

For insert query, comparing Fig. 5.4 with Fig. 4.6, the performance of the two cases is affected in the same way in *ObliviousDB* and *P-McDb*. That is, the execution time of both cases drops down when dividing the database into more groups, 'Case1' performs better than 'Case2', and the gap between them gets smaller when increasing the number of groups. The difference is, *P-McDb* takes much less time to process an insert query than *ObliviousDB* in both cases. In our test, *P-McDb* takes only around 0.0058 ms to encrypt and insert a record. When all the records are on one group, *P-McDb* takes 2.8 ms and 3.44 ms for running an insert query in 'Case1' and 'Case2', respectively.

5.5 Discussion

Although *P-McDb* protects the search, access, and size patterns from the CSPs, and achieves the forward and backward privacy. As discussed before, it also suffers from several limitations. In this section, we highlight the limitations of *P-McDb* and give the possible countermeasures.

The first issue of *P-McDb* is the collusion between the SSS and the WSS. In *P-McDb*, the SSS knows the search result for each query, and the WSS knows how the records are shuffled and re-randomised. If the SSS colludes with the WSS, they could learn the search and access patterns. To resist such collusion attacks, one possible solution is to generalise the system by introducing three or more CSPs, where one of them acts as the SSS and the others act as the WSS. In this case, the oblivious algorithm has to be performed with the cooperation of all the WSSs. Only when all the CSPs collude together, the search and access patterns will be leaked, which is significantly harder than the colluding between only two CSPs.

Moreover, in this work, we assume both the WSS and the SSS are honest. Yet, in order to learn more useful information, the compromised CSPs might not behave honestly as assumed in the security analysis. For instance, the SSS might not search all the records indexed by *IL*, and the WSS might not shuffle the records properly. Since both the WSS and the SSS are untrusted, *P-McDb* needs a mechanism to detect if both the SSS and the WSS honestly follow the designated protocols. We leave this problem to our future work. Last but not least, in [97], Stefanov *et al.* present a multi-cloud based ORAM system and give an approach to detect if the data is shuffled properly on the CSP. We can utilise their approach to *P-McDb*.

5.6 Related SE schemes Using multiple Servers

In the literature, there are several other SE schemes also use the multi-cloud strategy to minimise the sensitive information leakage or optimise the performance of the system.

In [97] Stefanov *et al.* present an ORAM storage system based on the multi-cloud strategy. By spreading the data and functionalities across two non-colluding CSPs, the access pattern is

hidden from the CSPs. In [64], Hoang *et al.* distribute the indices to two non-colluding CSPs and achieve the privacy of access pattern by re-encrypting and swapping the indices stored in the two CSPs after each query. In [41], Ishai *et al.* present a construction of a private outsourced database in the two-CSP model. By applying secure multiparty computation between the two CSPs, their solution hides all leakage due to access patterns between queries.

In [55], Bösch *et al.* introduce a distributed SE scheme that hides the search pattern. In their construction, the search operation is distributed across two non-colluding CSPs. By continuously re-shuffling the index as it is being accessed, none of the CSPs can tell which record was accessed and thus the search pattern of the scheme remains hidden. In [100], Orencik *et al.* propose an efficient scheme that allows privacy-preserving search over encrypted data using queries with multiple keywords. A two-CSP setting is employed to eliminate the correlation between the queries and matching files sent to the user under the assumption that the two CSPs are not colluding. Moreover, queries are randomised such that distinguishing two queries generated using the same search terms from unrelated queries is hard, *i.e.,* the search pattern is hidden.

The multi-cloud strategy is also used to improve the performance of the system. In [101], Kuzu *et al.* design a distributed secure index which allows parallel execution among multiple servers. In their study, the index is vertically partitioned and distributed to multiple servers to enable simultaneous decryption of large data payloads. By doing this, SE schemes can easily scale to big data. In [102], Poh *et al.* also propose a distributed SSE scheme over multiple servers. To prevent any one server from possessing a complete set of files or blocks of a file, their solution distributes files/blocks to many servers. Due to such distributions, their scheme hides the file sizes and the total number of files even after retrieval, while maintaining sublinear search time for each server.

5.7 Conclusion

In this work, we presented *P-McDb*, a dynamic searchable encryption scheme for multi-cloud outsourced databases. *P-McDb* does not leak information about the search, access, and size patterns. It also achieves both forward and backward privacy, where the CSPs cannot reuse cached queries for checking if new records have been inserted or if records have been deleted. Furthermore, *P-McDb* offers a flexible key management scheme where revoking users does not require regeneration of keys and re-encryption of the data. As future work, we plan to do our performance analysis by deploying the scheme in the real multi-cloud setting.

Chapter 6

Preserving Access Pattern Privacy in SGX-Assisted Encrypted Search

To process outsourced data efficiently and securely without sensitive information leakage, another promising practical approach to process outsourced data efficiently and securely is leveraging trusted hardware like Intel Software Guard Extension (SGX) [50]. SGX can isolate sensitive code and data in an encrypted memory region, called *enclave*. During execution, privacy and integrity of enclave memory are preserved with a set of hardware mechanisms. Recently, several SGX-based approaches for encrypted data access have been investigated in the literature. For instance, Fuhry *et al.* [103] present two constructions for sub-linear search over encrypted database by using SGX. Zheng *et al.* [105] introduce an SGX-assisted oblivious data analytics scheme that conceals the data access pattern. Moreover, several works [52, 106, 107] explore the deployment of ORAM on SGX. Unfortunately, these schemes suffer from several limitations. For instance, the constructions proposed in [103] leak the access pattern. Although the access pattern is concealed in [52, 106, 107], these schemes require long-term storage on SGX for managing the map between each data instance and its storage location, which is in large size when the database has millions of distinct values. These solutions are not very practical due to very limited memory resources in SGX. The scheme proposed by Zheng *et al.* [105] needs to scan the entire database linearly to answer a query. Another limitation is that there are side channel attacks when using SGX to ensure secure data access. Indeed, several recent works, such as [109–111], have shown that the Operating System (OS) can infer the data access pattern in SGX by launching side channel attacks, such as page faults, timing and cache attacks. State of the art solutions [52, 105–107] do not give the solutions to resist the side channel attacks. In Table 6.1, we compare existing SGX-based schemes.

Research Challenges. The main objective of this chapter is to preserve access pattern privacy by using SGX while ensuring efficient search over encrypted data and without any long-term

Table 6.1: Comparison of recent SGX-based schemes.

Scheme	Search complexity	Access pattern leakage	Side channel leakage	No long-term storage on SGX
Fuhry *et al.* [103] – Construction 1	$O(\log N)$	◐	○	✔
Fuhry *et al.* [103] – Construction 2	$O(\log N)$	○	○	✔
Gribov *et al.* [104]	$O(\log N)$	○	○	✔
Eskandarian *et al.* [52] – Linear	$O(N)$	●	○	✔
Eskandarian *et al.* [52] – Indexed	$O(\log^2 N)$	◐	○	✔
Zheng *et al.* [105]	$O(N)$	●	○	✔
Sasy *et al.* [106]	–	●	●	✘
Costa *et al.* [107]	–	●	●	✘
Ahmad *et al.* [108]	–	●	●	✘
SGX-assisted encrypted database [53]	$O(N/P)$	●	●	✔

○, ◐, and ● mean the information is completely leaked, partially leaked, and not leaked, respectively. ✔and ✘represent if long-term storage on SGX is not required or required, respectively. N represents the number of nodes in the tree or number of records in the database. P is page size in an SGX enclave. In [106, 107], no search operation is performed and this is denoted by –.

storage requirement in the enclave. As discussed below, there are two main challenges to achieve our objective: preserving the access pattern privacy while providing efficient search and withstanding side channel attacks.

To support efficient search, a straightforward method is to build indices and load only the required indices into SGX. However, this method leaks the *index access pattern* directly to the CSP, *i.e.,* the CSP can learn which indices match the query. To fully hide the index access pattern, a typical strategy is to load all the indices into SGX. Nonetheless, an SGX enclave only has around 90 MB memory for storing the code and data. For a large database, indices will exhaust the enclave memory and virtual memory mechanism of the OS. Consequently, this will significantly affect the performance of SGX. Moreover, as shown in [103], the actual pages that are accessed is still unprotected because of the page fault attack [109]. Therefore, the first challenge is to fully guarantee the access pattern privacy without exhausting the enclave memory when the database is large.

Although SGX provides a trusted environment for data processing, it suffers from side channel leakage, where the data is leaked when it is loaded into SGX [109–111]. Thus, the second challenge in our work is to protect the access pattern against potential side channel attacks. Several countermeasures, such as data oblivious access, balanced code execution, and data shuffling have been proposed in [112–114]. However, these techniques are too generic to be used in SE schemes. For instance, to defend against the page fault attack, Shinde *et al.* [113] propose to balance the code execution by adding and accessing dummy data. Nonetheless, we should ensure the added dummy data can be checked like real ones and do not affect the correctness of the search results, otherwise, the CSP can infer the real index access pattern.

Therefore, the techniques specific to SE schemes should be considered.

Our Contributions. In this chapter, we present an SGX-assisted SE scheme addressing the aforementioned research challenges. Basically, our solution uses the B+ tree structure to ensure search efficiency. To address the first challenge, our scheme loads and processes the tree nodes in batches. On the one hand, this method ensures that the index access pattern is protected. On the other hand, our scheme can process a large database without exhausting the enclave memory. Meanwhile, it protects the access pattern against the page fault attack. To mitigate other side channel attacks, *e.g.,* timing attacks, the B+ tree is searched in a balanced way, independent of the query and the access pattern. To analyse the performance, we evaluate our proposed scheme on Big Data benchmark [51] and compare it with ObliDB [52], an SGX-assisted ORAM-based database that protects the data access pattern. Our scheme outperforms ObliDB by at least 5.78× for range queries. We also compare our scheme with a baseline implementation without access pattern protection with sub-linear search support. There is always a tradeoff between security and performance. Our results show that our approach to protect the access pattern and defend side channel attacks introduces less than 27× overhead when the B+ tree contains around 1 million keys. Moreover, our approach does not require any long-term storage on the enclave. In summary, our contributions in this chapter are as follows:

- Our scheme protects the access pattern from the CSP by leveraging a trusted SGX.
- Our scheme prevents the CSP from inferring the access pattern by launching side channel attacks.
- Our scheme supports s simple equality match and complex operations including range, aggregate, and join queries.
- We have implemented a prototype of the system and tested its performance on an SGX-based hardware.

Chapter Organisation. The remainder of this chapter is organised as follows. First, Section 6.1 gives a brief overview of SGX functionalities and explains possible side channel attacks. An overview of our proposed solution is provided in Section 6.2. Then, we present our solution details in Section 6.3 and 6.4 before introducing its security analysis in Section 6.5. In Section 6.6, we report performance analysis. Section 6.7 reviews related work. Finally, we conclude this chapter in Section 6.8.

6.1 Background

6.1.1 Intel SGX

In this section, we give a brief introduction of SGX functionalities relevant to our system. For more details on SGX, we refer the reader to [50, 115]. SGX is an extension of x86 instructions for creating and managing software components, called *enclave*. Physically, the enclave is located inside a hardware guarded area of memory called Enclave Page Cache (EPC). The EPC consists of 4KB page chunks, and only around 90MB EPC can be used by the application. The SGX hardware enforces additional protection on the enclave, such that it is isolated from the code running on the system including the OS and the hypervisor.

Apart from the isolated code and execution, SGX has another two main security properties: sealing and attestation. Sealing is the process of encrypting enclave secrets for persistent storage to disk [116]. Every SGX processor has a key called the Root Seal Key that is embedded during the manufacturing process. Once an enclave is created, a seal key – which is specific to the enclave – is derived from the root seal key. When the enclave is torn down, this seal key is used to encrypt data and store the data in the disk. SGX also supports remote attestation, enabling a remote party to verify if an enclave is created properly on a trusted SGX. It also provides integrity to the code and data loaded into the enclave. Furthermore, the remote attestation feature helps in establishing a secure channel between an external party and the enclave.

6.1.2 Side Channel Attacks on SGX

Intel SGX offers secure execution environment by cryptographically protecting code and data on an untrusted server. Unfortunately, it is vulnerable to side channel attacks. As discussed below, there are at least three possible side channel attacks that could be launched by the untrusted server to derive sensitive information.

Page Fault Attack. An SGX program is executed in user mode, and it needs the underlying OS to manage virtual memory pages. Specifically, when launching an SGX process, the OS creates the page tables that map the virtual addresses to physical memory entries. However, when the virtual pages cannot be mapped to the physical memory, the CPU incurs a page fault and the faulting address will be reported to the OS. By manipulating the page table mappings, as shown in [109], a malicious OS can observe the page access pattern in SGX.

Timing Attack. Timing attack allows attackers to learn sensitive information by analysing the times taken to execute data-dependent operations. Every logical operation in a computer takes time to execute, and this time could differ based on the input, which is also the case for the operations in SGX. With precise measurements of the execution time for each operation, an attacker can backtrack the input data.

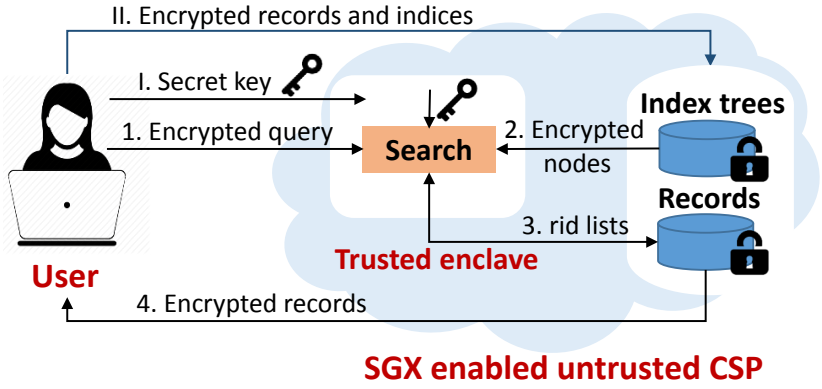


Fig. 6.1. An overview of the proposed approach.
The user shares the secret key with the trusted SGX enclave (Step I). The encrypted data and indices are stored in the untrusted cloud server (Step II). The query is decrypted and processed by the enclave (Steps 1, 2, and 3). After searching, the matched records together with a set of random records will be returned to the user (Step 4).

Cache Attack. Cache attack, or cache timing attack, is much more powerful, where an attacker can learn which cache lines are accessed by SGX. In the Intel X86 architecture, an L3 cache is shared among different CPU cores, while L2 and L1 caches are shared among different threads, including SGX threads. Thus, as discussed in [110,117], the classic Prime+Probe, Evict+Time, and Flush+Reload techniques are also effective to monitor the cache lines accessed by SGX.

6.2 Solution Overview

In this section, we first explain system entities and then describe the threat model and assumptions. Next, we briefly explain our design goals and the proposed approach. Finally, we give a security definition of our solution.

6.2.1 System Entities

Our system includes three entities: the user, SGX-enabled Cloud Service Provider (CSP) and an SGX enclave within the CSP. The users upload encrypted databases to the CSP and later issue encrypted queries and retrieve encrypted results. The CSP is responsible for storing encrypted data and loading data into the enclave for performing search. SGX enclave is responsible for processing user queries and returning search results to users without leaking any content and the access pattern to the CSP.

6.2.2 Threat Model and Assumptions

As done in other systems listed in Table 6.1, in our threat model, we assume the user is trusted. Similar to existing work (see Section 6.7), we assume adversaries could attack and fully control the OS running on the CSP, and they are curious about the data residing on the CSP. For simplicity, in the rest of the article, we regard the CSP as an adversary, which honestly follows the specified protocol but is curious to know the data. Since we employ encryption, the CSP is unable to access the data in cleartext. Moreover, the CSP cannot control and access the code and data within the enclave. Nonetheless, the CSP can interrupt the enclave as desired, by modifying the OS and SGX SDK, in order to learn side channel information. Therefore, we assume the CSP is able to exploit side channel attacks including the page fault attack [109], timing attack [118], and cache attack [117, 119] to infer the code paths and data access patterns inside the enclave. There are also other types of side channel attacks, such as branch shadowing [111], power monitoring [120], and electromagnetic [121], which are beyond the scope of this work.

SGX enclave is also considered to be trusted. In particular, both integrity and confidentiality of the code and data inside the enclave are protected with inherent cryptographic mechanisms. We also assume that SGX provides methods for establishing a secure channel with the users for protecting the communication between them.

6.2.3 Architecture Overview

Our system aims at enabling the CSP to process user queries in sub-linear time without leaking sensitive information about the content of the data and queries. Furthermore, we aim at protecting the access pattern of SGX against side channel attacks without exhausting the enclave memory even when the database is large. To achieve these goals, our basic idea is to encrypt the outsourced data and queries, build indices for the dataset, and enable SGX to perform the search operation in an oblivious manner.

The proposed approach is illustrated in Fig. 6.1. It consists of the trusted code inside the enclave for processing queries, the data storage on the CSP, and the encryption and decryption operations on the user side.

Initially, the user generates a secret key *sk* to encrypt the outsourced data and queries. The secret key *sk* is shared with SGX via a secure channel (Step I). To ensure search efficiency, the user first builds indices for the dataset and then uploads both the encrypted dataset and index trees to the CSP (Step II). For building an index, we use the B+ tree structure, the most popular one used in Database Management Systems (DBMSs) [103, 122]. When issuing a query, the user encrypts it using randomised encryption and *sk* (Step 1). Since *sk* is unknown to the CSP, the content of the query and whether the same query has been sent before (*i.e., search pattern*)

are protected from the CSP. With the secret key *sk*, the enclave decrypts the query and loads the associated index tree from the CSP for performing search (Step 2). To hide which nodes in the tree match the query, *a.k.a. the index access pattern*, the nodes are loaded and accessed in an oblivious manner that is independent of the query and could resist side channel attacks including page fault, timing, and cache attacks as explained in Section 6.3. After searching the B+ tree, SGX will know the identifiers *rid*s of the records matching the query predicate(s). The next step is to return the result to the user for select queries, or load and process the matched records for aggregate functions, such as 'MAX', 'SUM' and 'GROUP BY'. For both cases, to hide which records match the query, *a.k.a. the record access pattern*, SGX also returns or processes the matched records in an oblivious manner that could resist side channel attacks (Step 3). Finally, the user obtains the results in plaintext by decrypting them using *sk*.

6.2.4 Security Definition

The security of access pattern is defined as follows:

Definition 11 (Access Pattern Security). Let $\overrightarrow{H} := ((Q_1,R_1),\ldots,(Q_T,R_T))$ be the search history at time T, where Q_t denotes the query, and R_t is its search result at time t ($1 \leq t \leq T$). Let $A_I(\overrightarrow{H})$ and $A_R(\overrightarrow{H})$ be a sequence of accesses over the indices and the dataset of $\overrightarrow{H}$, respectively. We say that the access pattern is protected from the CSP if for any two search histories $\overrightarrow{H}_0$ and $\overrightarrow{H}_1$ of the same length, their access pattern $(A_I(\overrightarrow{H}_0),A_R(\overrightarrow{H}_0))$ and $(A_I(\overrightarrow{H}_1),A_R(\overrightarrow{H}_1))$ are computationally indistinguishable by the CSP.

6.3 Solution Detail

In this section, we first show how the data is represented and encrypted. Then, we describe how equality and range queries are processed in our system.

6.3.1 Data Representation

To support sub-linear search, given the database, the user first builds a standard B+ tree based on a defined branching factor. Formally, we define a B+ tree as $tree = \{b,L,N,\mathbf{cnt},\mathbf{nodes}\}$. Here, b is the branching factor, indicating each node can have up to b child nodes and $b-1$ keys. L is the number of levels of the tree, and the root node is in level $l = 1$. $\mathbf{cnt} = \{cnt_1,\ldots,cnt_L\}$ is an array of integers of length L. cnt_l represents the number of nodes in level l, where $l \in [1,L]$. The total number of nodes in the tree is $N = \Sigma_{l=1}^{l=L} cnt_l$. $\mathbf{nodes} = (node_0,\ldots,node_{N-1})$ is the array storing the N nodes in the tree. The order of storing nodes is from root to leave nodes and from left to right.

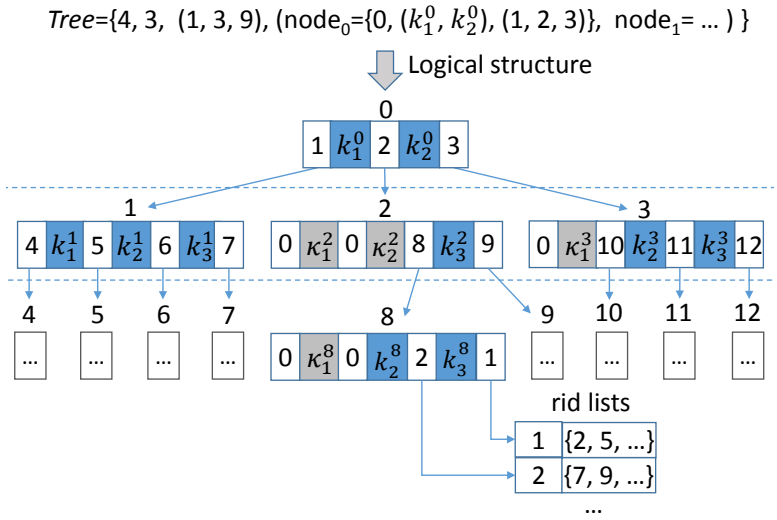


Fig. 6.2. A B+ tree index example with $branch = 4$, 3 levels, and 13 nodes. From root to leave nodes and from left to right, all the nodes are assigned a sequence of numbers in order as the *id*. Each non-root node stores 3 keys and 4 values, including both real and dummy keys and values. The dummy keys and values must be stored in the most left. The real value is the *id* of its child node. The dummy values are set to 0. The real value in each leaf-node is the *id* of a list of identifiers of the records matching the key.

The non-leaf node is defined as $node_{id} = \{id, \gamma, (k_1, \ldots, k_{b-1}), (cid_1, \ldots, cid_b)\}$. *id* is unique and used to identify the node, which also represents the node storage location in **nodes**. In the B+ tree, each node contains an array of keys $(k_1, \ldots, k_{b-1})$ for searching, and an array of its child nodes *id*s $(cid_1, \ldots, cid_b)$ for reaching its child nodes, where $node_i.cid_j = node_{cid_j}.id$. The $b-1$ keys in each non-root node consist of both real and dummy keys. On the one hand, the dummy keys are used to ensure all the nodes have the same size. On the other hand, the dummy keys are necessary to thwart side channel attacks (the reason will be given in Section 6.3.3). γ represents the number of dummy keys contained in the node.

The real keys are stored in the most right of the node and in increasing order, *i.e.,* $(k_1, \ldots, k_\gamma)$ and $(k_{\gamma+1}, \ldots, k_{b-1})$ are dummy and real keys, respectively, and $k_{\gamma+1} < \ldots < k_{b-1}$. The real keys separate the key domain into $b-\gamma$ subtrees that are reachable by $b-\gamma$ child node *id*s, *i.e.,* $(cid_{\gamma+1}, \ldots, cid_b)$. Moreover, the real keys in $node_{cid_i}$ are in $(k_{i-1}, k_i]$. Let $[\min, \max]$ be the domain for the keys. To ensure the dummy keys never match the query, we assign them the values out of the domain, *i.e.,* the values less than min or greater than max. Meanwhile, the dummy keys are not used to generate subtrees, so $(cid_1, \ldots, cid_\gamma)$ are set to 0.

The leaf node has the same structure as the non-leaf node. However, the leaf node does not have child nodes; instead, its cid_i points to a list of *rid*s of the records whose values are equal to k_i. Likewise, if the leaf node has less than $b-1$ real keys, a number of dummy keys will be

generated and stored in the most left of the key array. The record identifiers lists are encrypted and stored separately. Moreover, to hide size information, the user pads the lists to the same size before encrypting.

Fig. 6.2 illustrates the logical structure of a sample tree with $b = 4$, $L = 3$, and $N = 13$, where $cnt_1 = 1, cnt_2 = 3$, and $cnt_3 = 9$. The non-root node with less than 3 keys is padded with dummy keys that are inserted to the most left of the nodes. For instance, the first two keys κ_1^2 and κ_2^2 in $node_2$ and the first key κ_1^3 in $node_3$ are the dummy keys, and their cids are 0.

Index Access Pattern Protection. Our main objective is to hide the access pattern from the CSP. Protecting the access pattern means protecting both the index access pattern and record access pattern since the record and index are linked with each other. Protecting the index access pattern means protecting which nodes at each level are accessed from the CSP, excluding the root node. For each query, to hide its access pattern over the B+ tree, we need to answer three questions: **Question 1)** Which nodes should be loaded into SGX?; **Question 2)** Which loaded nodes should be searched?; and **Question 3)** Which keys should be checked in each searched node? In the following, we will give the answers to the three questions for equality, range and join queries.

Generally, to get the matched records in sub-linear time, SGX needs to search the root node in the first level and determine which nodes should be searched in the second level, and so forth. However, if SGX only loads the matched nodes at each level, the CSP can learn the index access pattern directly. To fully hide the index access pattern, for both equality and range queries, our solution to **Question 1** is loading the entire tree into SGX. In our approach, without exhausting the enclave memory, SGX reserves only one single EPC page and loads the tree in batches, rather than loading the entire tree in one go. Specifically, SGX loads and processes the tree nodes level by level. When the nodes at one level cannot be loaded into one EPC page, they will be grouped in fragments and loaded fragment by fragment, where each fragment contains the maximum number of nodes that can be loaded into one EPC page. Thus, our system can efficiently process the dataset in any size without exhausting the enclave memory. Moreover, which nodes are accessed within each loaded fragment is unknown to the CSP. Note that to load the indices from the CSP, the code execution context has to be switched between the trusted and untrusted environment, which is time-consuming. To reduce the frequency of context switching, SGX could also reserve more EPC pages and load more nodes per batch. For clarity, we focus on the case of reserving one EPC page and loading one fragment each time in the rest of this work.

6.3.2 Data Encryption

After building the tree, the user encrypts both the dataset and the index tree before sending them to the CSP. For encryption, given a security parameter λ, the user generates the secret key *sk*. Basically, each record is encrypted with *sk* using a randomised encryption algorithm such as AES-GCM. Each encrypted record is identified with a unique *rid* in plaintext.

In this work, to ensure the performance of our system, we do not aim to hide the tree structure from the CSP. That is, the CSP could learn the values of b, L, N and **cnt**. The user only encrypts the nodes in **nodes**. Specifically, the tree encryption is performed in three phases. In the first phase, all the nodes are padded to the same size as dummy keys and *cid*s. Assume all the nodes are s bits. It is known that the EPC page size is 4 KB, meaning each page can hold at most $P = \lfloor \frac{4KB}{s} \rfloor$ nodes.

In phase 2, the user fragments the nodes at each level P by P. In order to hide the order among the fragments, the user permutes the fragments with a pseudo-random permutation $\pi : \{0,1\}^{\lambda'} \times \{0,1\}^{\iota} \rightarrow \{0,1\}^{\iota}$. Each fragment has P nodes excluding the last fragment, such the nodes at each level will be loaded fragment by fragment. Afterwards, each fragment is encrypted with a semantically secure block encryption $Enc : \{0,1\}^{\lambda} \times \{0,1\}^{32768} \rightarrow \{0,1\}^{32768}$ ($32768 = 4KB$ is the data size in bit that can be hold in one EPC page), *e.g.,* AES-128 in GCM mode.

In the third and last phase, for each *cid* in each leaf node, the user first pads each *rid* list into a fixed size with dummy data, say -1, and then encrypts each list with *Enc* and *sk*. By padding the lists into the same size, the data distribution can be hidden from the CSP. Note that the list longer than the fixed size will be cut into multiple sub lists. In this case, the *cid* in each leaf node should contain multiple references pointing to the sub lists.

6.3.3 Searching Tree

When issuing a query Q, the user encrypts it by computing $EQ \leftarrow Enc_{sk}(Q)$, such the query is protected from the CSP. Moreover, *Enc* is semantically secure and search pattern is also hidden from the CSP.

The query is processed on SGX and the details are shown in Algorithm 10. Once received EQ, SGX first decrypts EQ with *sk* to get Q (Line 1), such it can learn the type of the query, the interested field(s) and keyword(s).

Tree loading. For different types of query, SGX loads the interested tree(s) in different ways. In this section, we focus on the equality and range queries, and give the details in lines 2 - 14. Specifically, SGX uses *mid* to cache the *id*(s) of the node(s) should be searched at the next level. When Q is an equality query, *mid* is set to be an integer of the same length as node *id* and initialised as *id* of the root node. Whereas, *mid* is set to be an N-bit string when Q is a

Algorithm 10 Query$(EQ, tree, P)$

1: $Q \leftarrow Dec_{sk}(EQ)$
2: **if** Q is equality query **then**
3: $\quad mid \leftarrow 0$
4: **if** Q is range query **then**
5: $\quad mid \leftarrow \{0\}^N$
6: $\quad mid[0] \leftarrow 1$
7: $Nodes[P] \leftarrow 0$ {Allocate an EPC for loading tree nodes}
8: **for** $l = 1$ to $l = tree.L$ **do**
9: $\quad$ **for** $i = 1$ to $i = \lceil \frac{tree.cnt_l}{P} \rceil$ **do**
10: $\quad\quad$ Load $Enc_{sk}(\mathbf{f}_i^l)$ to $Nodes$
11: $\quad\quad$ **if** Q is equality query **then**
12: $\quad\quad\quad EqualitySearch(Q, Nodes, tree.b, mid, l/tree.L)$
13: $\quad\quad$ **if** Q is range query **then**
14: $\quad\quad\quad RangeSearch(Q, Nodes, tree.b, mid)$

range query. Recall that N is the total number of nodes in the tree. The i-th bit in mid is used to mark if the i-th nodes in the tree is the matched one or not. Specifically, the id-th bit is set to 1 if the id-th node is the matched one, and 0 otherwise. Its first bit is initialised with 1 since the search operation starts from the root node. After searching the root node, SGX will know which nodes should be searched in level 2 and so forth.

For both equality and range queries, SGX reserves a fixed EPC page $Nodes$ (Line 7) and loads the required index tree(s) in batches for performing search (Line 8 - 14). Indeed, SGX loads and processes the permuted fragments at each level one by one. Since all the nodes at each level will be loaded, it is unnecessary to recover the order of the fragments before loading. In the following, we show how the loaded fragment is processed in detail for equality and range queries separately.

Solutions against Page Fault Attacks. Using a single EPC page to hold data blocks is not sufficient to resist the page fault attack. Indeed, if only the fragment containing the matched nodes (called *matched fragment* in short) is accessed, the CSP can still learn the index access pattern at page granularity by launching page fault attacks since only when the page is accessed the fault address will be reported to the OS. Our solution is to ensure every loaded fragment is accessed by SGX. Specifically, when the loaded fragment does not contain the matched nodes (called *unmatched fragment* in short) SGX searches several random nodes. In this case, even if the page fault exception occurs, the CSP is unable to know if it is caused by accessing the matched nodes or random ones.

Solutions against Timing and Cache Attacks. To defend against timing attacks, the time of searching the B+ tree should be independent of the query and the access pattern. Unfortunately, loading the tree in batches and searching random nodes in unmatched fragments are not sufficient to achieve this. Basically, there are still three issues making the index access pattern

vulnerable to timing attacks.

First, in a traditional B+ tree, the nodes may contain a different number of keys, and the time of processing a fragment depends on how many keys are checked in the accessed node. In turn, based on the processing time, the CSP can infer the number of checked keys, and then infer which node is accessed. For instance, assume checking one key takes T seconds. If the loaded fragment is processed in nT seconds, the searched node must contain no less than n keys. According to the size of each encrypted node, the CSP could infer which node is accessed with a high probability. To solve this issue, the real size of each node should be hidden after encrypting and our solution has achieved this by adding dummy keys.

The second issue is, when the same query is repeated, the fragments holding the matched nodes will be processed in fixed times, but the other fragments will be processed in variable times since the nodes to be searched in them are randomly picked and searching different nodes might take different times. By repeating the same query and detecting the processing time for each fragment, the CSP can infer the fragments with fixed processing times must be the matched ones, and vice versa. Thus, the CSP is still able to get the access pattern at page-level granularity. This issue can be prevented by appending timestamps to encrypted queries so that the CSP cannot replay issued queries.

However, even if all the encrypted nodes are of the same size and appended with timestamps, the CSP can still compare each fragment's processing time for different queries. For the queries searching the same tree, if each level has a fragment that is processed in the same time, there is a high probability that those queries are equivalent and the fragments processed in the same time contain the matched nodes.

Our scheme supports both the equality and range queries. Since the equality and range queries are processed in different manners, our solution takes different measures to prevent the timing attack. In traditional B+ trees, for each equality query, there is only one key that matches the query at each level, indicating only one node should be searched at each level. To protect the searched node from the page fault attack, SGX randomly searches an unmatched node for each loaded unmatched fragment. Therefore, for equality queries, to **Question 2**, our answer is always access one node in each loaded fragment. Furthermore, to defend against the timing attack, our solution is to ensure all the fragments are processed in constant time. Specifically, as mentioned in Section 6.3.1, the user pads all the non-root nodes into the same size by inserting dummy keys. Moreover, for each accessed node, SGX searches all the keys in the accessed node no matter if it is the matched one or a random one and no matter if the key is real or dummy, which is the answer to **Question 3** for equality queries. In this case, since all the nodes have the same number of keys and each key is processed in the same time, SGX processes each fragment in the same time for all equality queries.

For range queries, multiple keys could match the query, which means SGX might need

to access multiple nodes at each level. Moreover, it is hard to ensure all the fragments are processed in constant time for range queries since the number of nodes searched in each fragment varies with queries. Therefore, to withstand the timing attack, our idea is to ensure both the matched and unmatched fragments are processed at different times even if the same range queries are repeated. Basically, SGX searches a random number of nodes in unmatched fragments and also searches some random nodes when processing matched fragments. That is our answer to **Question 2** for range queries.

By mounting cache attacks, the CSP might still be able to learn how many and which nodes are searched in each fragment. However, if the CSP is unable to learn whether the searched nodes are matched or randomly picked ones, the index access pattern is still protected. Therefore, it is important to ensure the matched nodes and randomly picked nodes are indistinguishable for the CSP even if it mounts cache attacks. To achieve this, SGX searches the matched and randomly picked nodes in constant time and in the same manner. In Algorithms 11 and 12, we show how the loaded fragment is processed in details for equality and range queries, respectively.

6.3.4 Equality Query

Algorithm 11 EqualitySearch$(Q, Nodes, b, mid, isLastLevel)$

$Nodes \leftarrow Decrypt_{sk}(Nodes)$
$r_1 \xleftarrow{\$} \{0, len-1\}$, where len is the number of nodes in $Nodes$
$r_2 \leftarrow mid - Nodes[0].id$
if $Nodes[0].id \leq mid \leq Nodes[len-1].id$ **then**
    $node \leftarrow Nodes[r_2]$
    $flag \leftarrow 1$
else
    $node \leftarrow Nodes[r_1]$
    $flag \leftarrow 0$
$tid \leftarrow node.cid_b$
for $j = b-1$ to $j = 1$ **do**
    if $(Q.K \leq node.k_j$ **and** $isLastLevel = 0)$ **or** $(Q.K = node.k_j$ **and** $isLastLevel = 1)$ **then**
        $shift \leftarrow 1$
    else
        $shift \leftarrow 0$
    $tid \leftarrow tid - shift$
$mid \leftarrow mid * (1 - flag) + flag * tid$

For an equality query, only one node will be accessed in each loaded fragment, *i.e.,* the matched one or a random one. The challenge is how to ensure that the CSP is unable to learn whether the accessed node is the matched one or a random one via side channels.

***EqualitySearch*$(Q, Nodes, b, mid, isLastLevel) \rightarrow mid$.** The detail of processing each

loaded fragment is described in Algorithm 11, *EqualitySearch*. After searching the nodes at the upper level, the *id* of the matched node (*i.e., mid*) is cached in SGX. SGX first decrypts the loaded fragment (Line 1, Algorithm 11). Second, it checks if the matched node $node_{mid}$ is contained in the current fragment (Line 4). To hide if the accessed node is the matched one or a random one, the two cases are processed in the same way (Lines 2 - 9). Specifically, SGX pre-computes a random location r_1 and the possible location of the matched node r_2. If the matched node is included in this fragment, the r_2-th node in the page will be accessed. Otherwise, the r_1-th node will be accessed. Moreover, a $flag$ is used to mark if the searched node is the matched one or a random one. Specifically, $flag = 1$ when the searched node is the matched one, and $flag = 0$ otherwise.

Once the node to be searched is determined, the next step is to check which key inside the node matches the query. The enclave traverses all the keys inside the node (Lines 10 - 16). To resist the timing attack, the node should be processed in constant time. Thus, all the keys in the node should be checked in the same way no matter whether it is a dummy or real and whether it matches the query or not. Specifically, $Q.K$ is first assumed to be greater than the last key $node.key_{b-1}$ (Line 10). In other words, the last child node is assumed to be the matched one at the next level. From the most right to the left, SGX checks if each key $node.k_j$ is greater than or equal to the query. If yes, tid decrements, meaning the child node cid_{j+1} is not the matched one. Otherwise, tid is unchanged, indicating cid_{j+1} is the matched child node.

Nevertheless, the dummy keys in the searched node could also match the query since they are assigned with real values. To ensure the correctness of the search result, we should ensure the dummy keys cannot change the value of *mid*. As shown in Line 17, after searching a node, *mid* is updated based on two values: $flag$ and tid. When the searched node is randomly picked, $flag = 0$ and *mid* is not changed after searching no matter what tid is. On the contrary, when $flag = 1$, *i.e.,* the searched node is the matched one, *mid* will be updated with tid. We should ensure the dummy keys cannot change the value of tid, *i.e.,* $shift = 0$ and $Q.K$ is greater than all the dummy keys stored in matched nodes. Indeed, $Q.K$ should be greater than or equal to the first real key in the matched node. Recall that the dummy keys are stored in the most left of the node. By assigning the values out of the domain, they will never change tid since they are less than $Q.K$.

Note that, if the searched node is the matched leaf node, the result *mid* is the identifier of the matched *rid*s list.

6.3.5 Range Query

In our system, the predicate in range queries is parsed into $LK < x < RK$. In particular, LK (RK) is min (or max) value of the domain when the input predicate is $x < RK$ (or $LK < x$). The details for processing a range query are shown in Algorithm 12.

Algorithm 12 RangeSearch$(Q, Nodes, b, mid)$

1: $Nodes \leftarrow Decrypt_{sk}(Nodes)$
2: $r \xleftarrow{\$} \{0,1\}^{len}$, where len is the number of nodes in $Nodes$
3: **for** $i = 0$ to $i = len$ **do**
4: **if** $r[i] = 1$ **or** $mid[Nodes[i].id] = 1$ **then**
5: $node \leftarrow Nodes[i]$
6: **if** $Q.RK > node.k_{b-1}$ **then**
7: $mid[node.cid_b] \leftarrow mid[node.id]$
8: **else**
9: $mid[node.cid_b] \leftarrow 0$
10: **for** $j = b-2$ to $j = 2$ **do**
11: **if** $node.k_{j-1} < Q.LK \leq node.k_j$ **or** $node.k_{j-1} < Q.RK \leq node.k_j$ **or** $Q.LK \leq node.k_{j-1}$ **and** $node.k_j < Q.RK$ **then**
12: $mid[node.cid_j] \leftarrow mid[node.id]$
13: **else**
14: $mid[node.cid_j] \leftarrow 0$
15: **if** $Q.LK \leq node.k_1$ **then**
16: $mid[node.cid_1] \leftarrow mid[node.id]$
17: **else**
18: $mid[node.cid_1] \leftarrow 0$

***RangeSearch*$(Q, Nodes, b, mid) \rightarrow mid$.** SGX first decrypts the loaded fragment. To ensure the matched and randomly picked nodes are processed in a balanced way, SGX first precomputes a random bits string r (Line 2). Second, SGX traverses the EPC page (Lines 3 - 18), and searches the i-th node if $r[i] = 1$ or $mid[Nodes[i].id] = 1$. When $mid[Nodes[i].id] = 1$, the searched node is a matched one. Otherwise, it is a random one. Note that both r and mid are stored in registers and concealed from the OS. Thus, the OS is unable to learn the values of r and mid.

To ensure the matched and randomly picked nodes are processed in constant time and in the same manner, our answer to **Question 3** is that no matter if the searched node is a matched one or not, SGX traverses all its keys from the right to the left in the same way (Lines 6 - 14). More specifically, for each key $node.k_j$ $(1 \leq j \leq b-1)$, if there is a match, SGX assigns 1 to $mid[node.cid_j]$, such that when processing the nodes at the next level SGX knows $node_{cid_j}$ is a matched one. Otherwise, SGX assigns 0 to $mid[node.cid_j]$. Recall that the dummy keys in each node are either greater than max or less than min. By doing so, the dummy keys cannot match the query. Thus, the correctness of mid is guaranteed. Moreover, considering all the nodes have the same number of keys, all the accessed nodes will be processed in constant time no matter if they are the matched ones or randomly picked and no matter how many keys in them match the query.

6.3.6 Returning Search Results

After searching the tree index, SGX gets the identifier(s) of the matched *rid*s list. For select queries, the next step is to return the matched records to the user. The straightforward way is to send *mid* to the CSP. However, the CSP could learn the record access pattern.

In our system, the matched records are returned in an oblivious manner. Specifically, SGX first loads the matched list(s) together with a set of random lists, and which of them are the matched ones are unknown to the CSP. Second, to resist side channel attacks, SGX decrypts all the lists to get the *rid*s of the matched records. Third, the matched *rid*s and a set of random *rid*s are sent to the CSP, and the CSP sends the records identified with these *rid*s to the user. Fourth, SGX re-encrypts the matched *rid*s with *sk* to make it different from the one stored in the CSP, and sends it to the user, using which the user can identify which records are the matched ones. Finally, the user decrypts the matched records and get the data in plaintext.

Note that the CSP might learn the matched records by sending the same query to SGX repeatedly and comparing the returned records since the matched records are always the same but the unmatched ones are picked randomly. To avoid this issue, as mentioned before, the user can append a timestamp to each encrypted query.

Algorithm 13 MAX$(Ercds, rids, m)$

```
1: max ← MIN_m
2: for i = 1 to |Ercds| do
3:     rcd ← Dec_sk(Ercds[i])
4:     if rcd.id ∈ rids then
5:         flag = 1;
6:     else
7:         flag = 0;
8:     if (rcd.D_m − max) ∗ flag > 0 then
9:         t = 1
10:    else
11:        t = 0
12:    max ← rcd.D_m ∗ t + max ∗ (1 − t)
13: return max
```

6.4 Complex Queries

Our system also supports the aggregate and join queries. In this section, we give the details to secure the access pattern for those complex queries.

Algorithm 14 SUM$(Ercds, rids, m)$

```
1: sum ← 0
2: for i = 1 to |Ercds| do
3:     rcd ← Dec_sk(Ercds[i])
4:     if rcd.id ∈ rids then
5:         flag = 1;
6:     else
7:         flag = 0;
8:     sum ← sum + rcd.D_m * flag
9: return sum
```

6.4.1 Aggregate Functions

For aggregate queries, such as 'MIN', 'MAX', 'SUM', and 'AVG', the matched records have to be loaded into SGX for further computation, rather than returned to the user. To hide the record access pattern, the matched records should be concealed from the CS. Like select queries, for aggregate queries, our solution is to load a set of unmatched records in addition to the matched ones to SGX. In this case, from the view of the CS, each loaded record could be the matched one or a random one with 50%. The challenge is to prevent the CS learning which records are the matched ones by mounting side channel attacks and ensure the correctness of the returned aggregate value at the same time. The main idea of our solution is to process all the loaded records in the same manner and constant time. In Algorithms 13 and 14, we take the 'MAX' and 'SUM' as examples to show how our solution ensures both security and correctness, respectively.

For both 'MAX' and 'SUM' functions, our algorithms take the loaded encrypted records, including both the matched and random ones, the matched *rid*s list and the field identifier m as the inputs. SGX first decrypts each record. Note that it is unnecessary to decrypt the unmatched records for getting the 'MAX' and 'SUM' values, yet it is necessary to ensure each loaded record is processed in constant time. Second, SGX detects if each record is the matched one or not by checking if its *id* is in the matched *rid*s list, and this can be implemented by building a hash table for the matched *rid*s. The matched and random records are marked with $flag = 1$ and $flag = 0$, respectively. Instead of skipping the unmatched records, the m-th value of both the matched and unmatched records are multiplied by $flag$ and counted to the aggregate functions (Line 8 in Algorithm 13 and Line 8 in Algorithm 14). Due to $flag = 0$ for unmatched records, counting unmatched records does not affect the correctness of the aggregate functions. Moreover, by processing the matched and unmatched records in the same way, the CS is unable to distinguish them by mounting page fault, timing and cache attacks. Recall that the replay attack can be blocked by appending a timestamp to each encrypted query.

6.4.2 Join Queries

Algorithm 15 Join($tree_1, tree_2$)

1: $res \leftarrow \emptyset$
2: $Nodes_1[P] \leftarrow 0, Nodes_2[P] \leftarrow 0$
3: **for** $i = 1$ to $\lceil \frac{tree_1.cnt_L}{P} \rceil$ **do**
4: Load $tree_1.Enc_{sk}(\mathbf{f}_i^L)$ to $Nodes_1$
5: $Nodes_1 \leftarrow Decrypt_{sk}(Nodes_1)$
6: $len_1 \leftarrow$ Number of keys in $Nodes_1$
7: **for** $j = 1$ to $\lceil \frac{tree_2.cnt_L}{P} \rceil$ **do**
8: Load $tree_2.Enc_{sk}(\mathbf{f}_i^L)$ to $Nodes_2$
9: $Nodes_2 \leftarrow Decrypt_{sk}(Nodes_2)$
10: $len_2 \leftarrow$ Number of keys in $Nodes_2$
11: $JoinSearch(Nodes_1, len_1, Nodes_2, len_2, res)$
12: **return** *res*

Algorithm 16 JoinSearch($Nodes_1, len_1, Nodes_2, len_2, res$)

1: $i = 1, j = 1$
2: **while** $i \leq len_1$ **and** $j \leq len_2$ **do**
3: **if** $Nodes_1.k_i = Nodes_2.k_j$ **then**
4: $t_1 \leftarrow 1, t_2 \leftarrow 1$
5: **else if** $Nodes_1.k_i > Nodes_2.k_j$ **then**
6: $t_1 \leftarrow 0, t_2 \leftarrow 1$
7: **else**
8: $t_1 \leftarrow 1, t_2 \leftarrow 0$
9: $res \leftarrow res \cup Nodes_1.cid_i * t_1 * t_2 \cup Nodes_2.cid_j * t_1 * t_2$
10: $r \xleftarrow{\$} \{1,2\}, t_r = 1 - t_{3-r}$ {*i.e.*, $t_1 = 1 - t_2$ or $t_2 = 1 - t_1$}
11: $i = i + t_1, j = j + t_2$

A join query is a means for combining columns from one (self-join) or more tables by using values in related fields common to each. Nested loop join [123], Hash join [124] and sort-merge join (*a.k.a.* merge join) [125] are the common join algorithms used in variable DBMSs. Unfortunately, they cannot be utilised to our system directly because of the leakage of access pattern.

Tree loading. In this work, we focus on the join between two tables and implement the join query by finding identical keys in the related B+ trees. Considering all the keys incur in the database are stored in the leaf nodes of the related tree, SGX only searches the leaf nodes of the joined trees. Therefore, hiding the index access pattern of join queries means hiding the location of identical keys. To achieve this, we also need to answer the three questions given in Section 6.3. Our answer to **Question 1** is loading all the leaf nodes in the two trees fragment by fragment. By doing so, the CS cannot learn the index access pattern by observing the loaded nodes. The details are provided in Algorithms 15 and 16. Specifically, SGX reserves two EPC

pages $Nodes_1$ and $Nodes_2$ (Line 2, Algorithm 15), and loads each pair of fragments in the two joined trees (Lines 3-11, Algorithms 15). Moreover, to thwart the page fault attack, each loaded fragment should be accessed. Thus, SGX decrypts each loaded fragment (Line 5 and 9). After decrypting, SGX also gets the total number of keys, *e.g.,* len_1 and len_2, included in the two loaded fragments. In this section, for clarity, we use $Nodes.key_{j*(b-1)+i}$ to present the i-th key in $Nodes[j]$, where $i \in [1, b-1]$ and $j \in [1, P]$.

Node searching. Considering the keys in each fragment are stored in order, the sort-merge join algorithm is the most appropriate method to find the identical keys. However, the original sort-merge algorithm is susceptible to timing and cache attacks. According to the original sort-merge join algorithm, SGX first compares $Nodes_1.key_1$ with $Nodes_2.key_1$. When $Nodes_1.key_1 = Nodes_2.key_1$, $Nodes_1.key_2$ and $Nodes_2.key_2$ will be loaded into the CPU for the second round comparison; when $Nodes_1.key_1 > Nodes_2.key_1$, $Nodes_2.key_2$ will be loaded and compared with $Nodes_1.key_1$; when $Nodes_1.key_1 < Nodes_2.key_1$, $Nodes_1.key_2$ will be loaded and compared with $Nodes_2.key_1$; and so forth. In other words, the relationship between the two searched keys determines which and how many keys should be loaded for the next round of comparison. Furthermore, which and how many keys are loaded for each round of comparison affects the process time of each fragment pair and the state of the cache lines. For instance, assume $len_1 \leq len_2$, if there are len_1 identical keys in the two fragments, two keys will be loaded each time for the first len_1 rounds of comparison, and SGX just needs to perform len_2 rounds of comparison in total. On the contrary, if there is no identical key in the two fragments, only one key will be loaded for each round of comparison (excluding the first round), SGX has to perform $len_1 + len_2$ rounds of comparison. Thus, the time of searching each pair of fragments depends on the number of identical keys they contained. By mounting timing attacks, the CS can infer how many keys are identical in each pair of fragments. Even worse, by mounting cache attacks, the CS can learn the exact locations of the identical keys, since two keys will be loaded into the cache lines when there is a match.

To protect the index access pattern from the timing and cache attacks, we modify the sort-merge algorithm. Our main idea is that no matter if there is a match or not between the two compared keys, SGX always only loads the next key of either $Nodes_1$ or $Nodes_2$ for the next round of comparison. In this case, no matter how many keys are identical, the two fragments always need $len_2 + len_1$ rounds of comparisons, making the process resistant to timing attacks. Moreover, no matter if each pair of compared keys are identical or not, SGX only loads one new key into the cache lines for the next round of comparison. As a result, the CS cannot infer the index access pattern via cache attacks. In short, our answer to **Question 2 and 3** is searching all the keys in each pair of fragments. The details of our solution are given in in Algorithm 16, *JoinSearch*.

Basically, *JoinSearch* takes two fragments as input, and outputs the *cid*s of the matched

keys between the two fragments. SGX uses two flags t_1 and t_2 to mark the relationships between the checked two keys $Nodes_1.k_i$ and $Nodes_2.k_j$ (Lines 3 - 9). Specifically, if $Nodes_1.k_i = Nodes_2.k_j$, $t_1 = 1$ and $t_2 = 1$. If $Nodes_1.k_i > Nodes_2.k_j$, $t_1 = 0$ and $t_2 = 1$. If $Nodes_1.k_i < Nodes_2.k_j$, $t_1 = 1$ and $t_2 = 0$. In any case, $Nodes_1.cid_i * t_1 * t_2$ and $Nodes_2.cid_j * t_1 * t_2$ will be added into the result *res*. However, only when $t_1 = 1$ and $t_2 = 1$, the *cid* is really added into *res*. By doing so, SGX can process the three cases in the same manner while ensuring the correctness of the result. In the next loop, $Nodes_1.k_{i+t_1}$ and $Nodes_2.k_{j+t_2}$ will be compared. Thus, which key will be loaded depends on if $t_0 = 1$ or $t_1 = 1$. To avoid loading two new keys when $Nodes_1.k_i = Nodes_2.k_j$, either t_1 or t_2 is turned into 0 (Line 10). As a consequence, when $Nodes_1.k_{i+1}$ is loaded, the CS can learn $Nodes_1.k_i = Nodes_2.k_j$ or $Nodes_1.k_i < Nodes_2.k_j$. Otherwise, the CS can learn $Nodes_1.k_i = Nodes_2.k_j$ or $Nodes_1.k_i > Nodes_2.k_j$. In both cases, the CS can only successfully guess the relationship between the $Nodes_1.k_i$ and $Nodes_2.k_j$ with 50%.

6.5 Security Analysis

In this section, we first analyse the leakage in our scheme and then prove that our scheme protects the access pattern from the CSP.

Leakage. To ensure the performance of the system, in this work, we do not make any effort to hide the tree structure and database size. Basically, for each B+ tree[1], the number of its levels, the number of nodes at each level, and the size of each encrypted node are leaked to the CSP. Moreover, from the search history, the CSP knows which tree is searched for each query, and the length of each encrypted query. For the dataset, the number of records and the size of each encrypted record are also leaked. Such information leakage could be minimised by introducing dummy records and padding the queries and records into the same sizes. However, dummy records and padding incur overheads, thus reducing the performance of the system. In the following, we prove our scheme ensures the security of the access pattern for the queries with the same type, the same number of predicates and searching over the same fields.

Theorem 6.1. *Let* $\alpha = \frac{Number\ of\ returned\ records}{Number\ of\ matched\ records}$ *for select queries, and* $\alpha = \frac{Number\ of\ loaded\ records}{Number\ of\ matched\ records}$ *for aggregate queries. When* α *is big enough, the proposed system protects the access pattern from the CSP for range, aggregate, and join queries.*

Proof. Let $\overrightarrow{H}$ be the query history of size T. In the following, we justify that both the index and record access patterns are protected from the CSP for equality, range, aggregate and join queries.

[1]There is a B+ tree for every field in the database.

Equality and Range queries. In our system, for both equality and range queries, SGX always loads the whole tree into the enclave in batches, *i.e.,* fragment by fragment. That is, the index access pattern for each equality or range query is always the whole tree from the CSP perspective. Moreover, after each loading, which node is accessed inside the loaded fragment is invisible to the CSP. Therefore, without mounting side channel attacks, the index access pattern for equality and range queries is protected from the CSP.

We prove the index access pattern is also protected even when the CSP mounts page fault, timing, and cache attacks. Recall that by mounting page fault attack the CSP can only learn whether the loaded fragment is accessed or not, yet it cannot learn which nodes within the fragment are accessed, since the CSP OS can only learn a 4 KB-granular page address [50,103]. In our solution, no matter if the loaded fragment contains the matched nodes or not, SGX always searches some nodes. In this case, when mounting page fault attacks, the CSP always gets the fault report no matter which fragment is being processed. However, the CSP cannot tell if the fault report is caused by accessing the matched nodes or random nodes. Thus, the index access pattern at page granularity is also protected from the CSP.

In our solution, each node has the same number of keys, and all the keys in the matched node are searched in the same way for both the equality and range queries. For equality queries, only one node is searched in each fragment. Therefore, for equality queries, each fragment is processed in constant time no matter which node inside it is searched. Thus, the index access pattern for equality queries is also protected against the timing attacks. For range queries, the number of searched nodes in each fragment is randomised sine SGX always search several unmatched nodes randomly. Consequently, the time of processing each fragment is also random. However, no matter which nodes are searched, each node will be searched in constant time, since they all have the same number of keys and processed in the same way. Hence, based on the processing time for each fragment, the CSP can learn how many nodes are searched, but it cannot infer which nodes inside the fragment are searched.

By mounting cache attacks, the CSP might learn which nodes are searched inside each fragment for equality and range queries. However, it cannot tell which of them are the matched ones since all the nodes are searched in an oblivious manner. Therefore, the index pattern is also protected from cache attacks.

After searching the B+ tree, the next step for equality and range queries is returning the matched records to users. While returning records to users, the CSP sees $A_R(\overrightarrow{H})$, which is a sequence $(R'_1,\ldots,R'_T)$, where each R'_t consists of the matched records R_t and a set of unmatched records. All the records stored in the CSP are encrypted using randomised encryption. Therefore, the records in each R'_t are indistinguishable from a set of random bit strings by the definition of randomised encryption. Note that although R'_t is revealed to the CSP, it cannot differentiate matched records from the random ones, and it can infer the precise access pattern

with $\frac{1}{2}^{|R'_t|}$ probability, where $|R'_t| = \alpha|R_t|$. By increasing the value of α, $\frac{1}{2}^{|R'_t|}$ can be negligible.

Another issue is if the CSP can infer search pattern from $A_R(\overrightarrow{H})$. Indeed, when $Q_i = Q_j (1 \leq i, j \leq T)$, the matched records must be the same, so that $R'_i \cap R'_j \neq \emptyset$. However, when $Q_i \neq Q_j (1 \leq i, j \leq T)$, if CSP returns the same unmatched records to the user, $R'_i \cap R'_j \neq \emptyset$. Since the encrypted queries are semantically secure, the CSP cannot tell whether the queries involve the same keywords or not. Thus, the CSP cannot tell the intersection between any two results are the matched or unmatched records. Only when $R'_i \cap R'_j = \emptyset$, the CSP can learn $Q_i \neq Q_j$. On the contrary, it cannot determine if they are equal or not when $R'_i \cap R'_j \neq \emptyset$.

Aggregate query. The main difference between aggregates and select queries is that the matched records are loaded into SGX for further processing rather than returned to the user directly. Therefore, we just need to prove that the way to process the aggregate does not leak the record access pattern to the CSP. Our solution takes two measures to protect the record access pattern for aggregate operations. First, SGX loads a set of unmatched records to mask the matched ones for further processing. Second, the matched and unmatched are processed in a balanced way, where the matched and random records are processed in the same way without affecting the correctness of the aggregate function. By doing so, the CSP cannot tell whether each record is a matched one or not even when mounting side channel attacks. Therefore, whether the CSP can infer the records access pattern depends on the value of α, *i.e.,* the ratio between loaded and matched records. As mentioned above, the probability of guessing the records access pattern can be negligible by increasing the value of α.

Join query. For join queries, SGX also either returns the records to users or loads the matched records for further process. In both cases, we have justified that the record access pattern is protected from the CSP above. Therefore, we just need to justify the index access pattern is protected from the CSP.

In our solution, for join queries, SGX always loads all the fragments into the last level of the joined trees. That is, just via the manner of tree loading, the CSP cannot learn which leaf nodes are searched by SGX. Like equality and range queries, SGX also accesses each pair of fragments no matter if they contain identical keys or not. Thus, the CSP cannot learn the index access pattern by mounting page fault attacks. Moreover, as described in Section 6.4.2, each pair of fragments are processed in the same manner and constant time. Therefore, the index access pattern of join queries is also protected from timing and cache attacks.

6.6 Performance Analysis

In this section, we demonstrate the performance of our scheme.

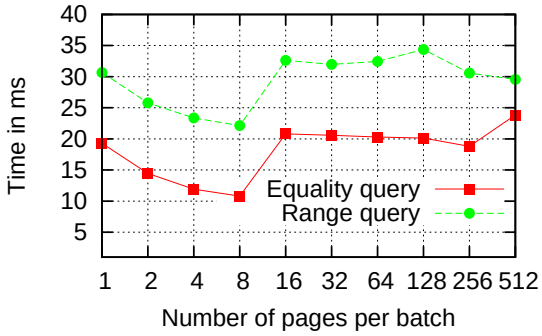


Fig. 6.3. Tree search time with 1 million keys. The branch factoring=32.

Table 6.2: The details of the tested queries.

	Query
Q_1	Select * from ORDERS where 1506 < O_ORDERKEY < 2016
Q_2	Select * from RANKINGS where pageRank > 12000
Q_3	Select * from RANKINGS where pageRank > 1000
Q_4	Select COUNT(*) from CFPB where (Product="Credit card" or Product= "Mortgage") and Timely_Response="No" GROUP BY Bank
Q_5	select sourceIP, totalRevenue, avgPageRank from (select sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue from Rankings as R, UserVisits as UV where R.pageURL = UV.destURL AND UV.visitDate between Date('1980-01-01') and Date('1980-04-01') GROUP BY UV.sourceIP) ORDER BY totalRevenue DESC LIMIT 1

6.6.1 Implementation

We implemented and evaluated the performance of our system on an Intel NUC 7i3BNH, with a 4-core Intel i3-7100U 2.4GHZ processor with SGX enabled and 8GB RAM. The prototype of the proposed system is implemented in C/C++. The cryptography primitives used on the user side, such as the records and indices encryption, are implemented based on Libgcrypt 1.8.2 library [126]. The trusted code on SGX is implemented based on SGX SDK 2.0. Specifically, we use 'sgx_rijndael128GCM_decrypt' to recover the plaintext of loaded nodes in SGX. The performance is tested on real SGX hardware. All the times presented in the following are averaged over 100 runs.

6.6.2 TPC-H Benchmarking

We first evaluated the performance of our scheme with TPC-H [86] dataset. The table used in our experiment is the 'ORDER' table, which consists of 1.5 million records and 9 fields. We built a B+ tree index for 'O_ORDERKEY' field consisting of 1032331 keys (29 of them are dummy keys).

Table 6.3: Comparison with ObliDB and the baseline.

Data Set	Keys in B+ Tree	#Pages /Batch	Query	Result Size	Our Work	ObliDB Indexed	Speedup	Baseline	Overhead
Range queries									
ORDERS	1032331	8	Q_1	125	22.15ms	–	–	0.815ms	27.2×
RANKINGS	1143	1	Q_2	1	0.37ms	2.14ms	5.78×	0.072ms	5.14×
RANKINGS	1143	1	Q_3	88	0.37ms	126ms	340.5×	0.077ms	4.81×
Aggregates and Joins									
CFPB	9	1	Q_4	–	1.67ms	0.869s	520.4×	1.5ms	1.11×
UNIVERSITIES	20631	1	Q_5	91	0.466s	19.56s	4.79×	0.455s	1.02×

6.6.2.1 Equality and Range Queries

For a traditional B+ tree, the search complexity is $O(\log_b N)$, where N represents the number of nodes and b is a branching factor of the tree. However, in our system, in order to resist side channel attacks, the B+ tree is accessed in an oblivious manner, where the nodes searched in each fragment could be those matching the query or randomly picked. Specifically, within every P nodes, at least one of them is accessed. Thus, the search complexity for equality and range queries is $O(\frac{N}{P})$ in our system.

To evaluate the performance of equality and range queries, we set the branch factoring to 32 and evaluated the B+ tree search time with different batch sizes, *i.e.,* the number of fragments loaded per batch, which is also the number of EPC pages reserved in the enclave. We tested the B+ tree searching time for both the equality and range queries by changing the batch size from 1 to 512. The result is shown in Fig. 6.3.

From Fig. 6.3, first we can see that the search time of executing an equality query is much less than executing a range query. That is because only one node should be searched in each page for equality queries, however, multiple nodes should be searched for range queries. Second, we see the searching time goes down when loading more pages each time, and reaches the lowest point when loading 8 pages per batch, which are about 10.8 milliseconds (ms) and 22.15 ms for equality and range queries, respectively. However, when loading more than 8 pages, the searching time goes up again. That is because there is less context switching between the untrusted and trusted code when loading more nodes in each batch. However, when loading more than 8 pages, the enclave memory is exhausted, and part of the data has to be swapped in and out between the enclave and the disk, which is significantly costly.

We also implemented the second construction (*i.e.,* Construction 2) of HardIDX [103] as a baseline, where only the matched nodes in the B+ tree are loaded into SGX for searching. Comparing with the baseline, our scheme takes several strategies to protect the access pattern from the CSP. In Table 6.3, we compare the search time of a simple range query (*i.e.,* Q_1 in Table 6.2) with the baseline case. When loading 8 pages per batch, the strategies taken in our scheme to protect the index access pattern incurs 27.2× performance overheads for range

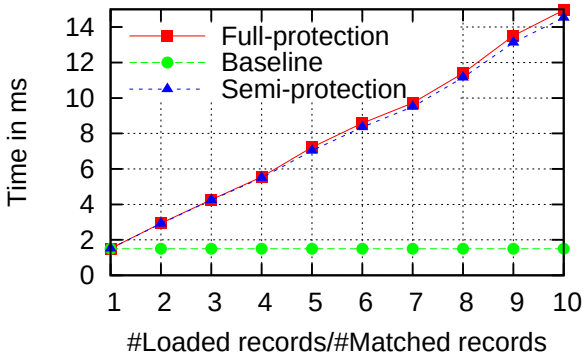


Fig. 6.4. The time of finding the MAX value among 1000 matched records.

queries.

6.6.2.2 Aggregate Queries

Once the search operation is finished, the next step for the queries with aggregate functions is to load the matched records into SGX and do further computations. When loading the matched records into SGX, the records access pattern should also be hidden from the CSP. Otherwise, the CSP can still mount leakage-based attacks. Our scheme takes two measures to protect records access pattern from the CSP. First, SGX loads a set of random records apart from the matched ones, such that the matched ones are blinded. As discussed in Section 6.5, the probability of inferring the access pattern is affected by the number of loaded random records. Roughly speaking, the more records loaded, the more difficult for the CSP to infer the access pattern. However, the more records loaded also means the more work is expected to be done on SGX. In the second experiment, we tested the time of processing aggregate functions with different α values. Second, to ensure the CSP cannot infer if each loaded record is a random one or a matched one by mounting side channel attacks, SGX also processes the records in a balanced way, where the matched and random records are processed in the same way without affecting the correctness of the aggregate function.

In our experiments, we considered the 'MAX' aggregate function as an example to show the processing overhead on SGX. Moreover, we compared the performance of our scheme with a baseline case where SGX only loads matched records, and a *semi-protection* case where SGX also loads random records but ignores side channel attacks. We call the solution taking the above two measures as *full-protection*. For the test, a query with 1000 matched records was tested over the 'ORDER' table. The test result is shown in Fig. 6.4. First, Fig. 6.4 shows that the processing times of both the full-protection and semi-protection cases increase linearly with α. That is, the more random records loaded, the more computation is needed on SGX,

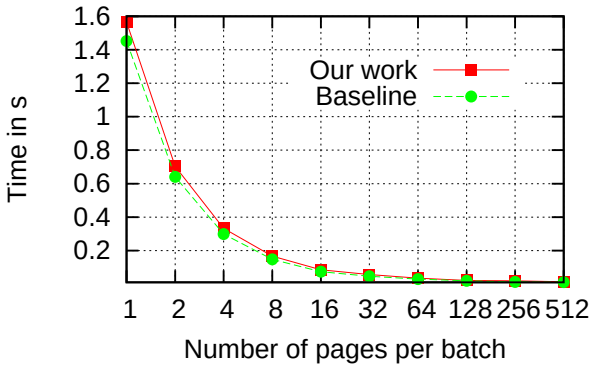


Fig. 6.5. The time of executing a join query.

which also ensures the higher security level of the access pattern. Moreover, the two lines almost overlap, which means the overhead to thwart side channel attacks is much less the one to hide the record access pattern.

6.6.2.3 Join Queries

To evaluate the performance of join queries, we tested a join query between 'ORDERS' and 'CUSTMERS' tables by changing the batch sizes. The tested query is 'select ORDERS.O_ORDERDATE, CUSTOMER.C_NAME from ORDERS inner join CUSTOMER on ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY', which returns 99933 matched records. The 'CUSTOMER' table contains 150,000 different 'C_CUSTKEY' values.

To better show the overhead increased by the index access pattern protecting techniques, we also compared the performance of our solution with a baseline case, where only the fragments containing matched keys are searched. The test result is shown in Fig. 6.5.

Fig 6.5 shows that the query execution time decreases with batch size, which is different from the case of equality and range queries. The reason is that, in equality and range queries, the search time is significantly less than other operations, such as the context switching between trusted and untrusted code and memory managing. However, in join queries, the searching time is much higher than others, since much more keys should be searched. In the tested join queries, around 70% time is taken by searching the keys. Although the data being swapped in and out between the enclave and the untrusted memory when loading more than 8 pages per batch, the search time is getting better. Thus, the overall executing time of the join query decreases with batch size even when loading more than 8 pages per batch.

Fig. 6.5 also shows that the gap between our approach and the baseline gets closer when increasing the number of pages loaded each batch. They almost take the same time when loading more than 128 pages per batch.

6.6.3 Big Data Benchmarking

ObliDB [52] is one of the ORAM-based solutions that can also protect the access pattern by using SGX. In [52], Eskandarian *et al.* have shown that their indexed solution is much more efficient than Opaque [105]. In the following, we will show our scheme is much more efficient than ObliDB indexed solution.

For the comparison, as done in ObliDB, we evaluated our scheme with the datasets and queries in Big Data Benchmark [51]. Specifically, we tested two range queries on two 'RANKINGS' tables, and two complex queries with aggregates and joins operations on 'CFPB', 'RANKINGS', and 'USERVISITS' tables. The tested queries and the information of the tested datasets and are shown in Table 6.2 and 6.3, respectively.

We downloaded the code of ObliDB from [127] and tested its performance on our own machine. Moreover, we also tested the baseline implementation, where only the matched nodes are loaded into SGX for search, with the same queries and datasets. For testing our scheme, the branching factor was fixed to 4, only 1 page was loaded per batch, and the ratio α was set to be 2. Note that our scheme will perform better when loading 8 pages per batch.

The test results for our scheme, ObliDB, and the baseline are shown in Table 6.3. For range queries, in our scheme, both Q_2 and Q_3 took only 0.37 ms, including both the B+ tree search time and the *rid*s list processing time. However, ObliDB is built based on ORAM, making its performance sensitive to the result size. When the result sizes are 1 and 135, ObliDB took 2.14 ms and 216 ms, respectively. Our scheme achieves at least 5.78× speedup than ObliDB indexed solution for range queries. Comparing with the baseline, our solution increases around 5× overhead.

For aggregates and joins, our scheme also outperforms ObliDB significantly. Specifically, for the query with 'GROUP BY' and 'COUNT', *i.e.,* Q_4, our scheme is over 520× faster than ObliDB, and introduces only 1.11× overhead when compared with the baseline. For the join query Q_5, our scheme outperforms ObliDB by 4.79× and is 1.02× slower than the baseline.

6.7 Related Work

In [103], Fuhry *et al.* present two SGX-assisted constructions for search over encrypted data. To support sub-linear search, the B+ tree index is utilised in both constructions. In the first construction, the encrypted index tree is entirely loaded into the enclave for performing search. When the index size is large, the enclave pages have to be swapped in and out due to the enclave memory limitation, which affects the performance significantly. Moreover, the untrusted server could observe data access inside the enclave with a page-level granularity by leveraging the page fault side channel attack [109]. In their second construction, only the nodes involved in

the tree traversal are loaded into the enclave. In this case, the accessed nodes are leaked to the CSP directly.

In [104], Gribov *et al.* also present a B+ tree-based SGX-assisted encrypted database, fully supporting SQL queries, called *StealthDB*. They also reduce the context switching overhead between the enclave and the untrusted server memory by $5\times - 10\times$ by using an exit-less communication mechanism [128]. Unfortunately, StealthDB still leaks the index access pattern since it only loads the matched nodes into the enclave for performing a search operation.

Eskandarian and Zaharia [52] also proposed two basic methods for the data storage and access, named linear and indexed storage. For linear storage, SGX searches each record one by one and then loads the matched records with ORAM primitives. This method can conceal the access pattern effectively, but it incurs significantly high computation overheads. In the indexed storage, a B+ tree is searched first to narrow down the records to be scanned. However, the access pattern over the B+ tree is leaked.

Zheng *et al.* [105] study how to leverage SGX to secure distributed analytical workloads, and propose a system called *Opaque*. By sorting and shuffling the data, Opaque could avoid the access pattern leakage. However, Opaque linearly scans and sorts the entire database to answer a query, which is inefficient for large databases. Further, they do not give the solution to address side channel attacks, from which the attacker could still infer sensitive information.

In [106], Sasy *et al.* design and implement a library of ORAM primitives running on SGX that can defend against side channel attacks. However, in their design, the enclave has to store a map between each distinct value and its storage location. Due to the limited memory of SGX enclave, their implementation is not scalable to the databases with a large number of distinct values. In particular, when each distinct value in the database only occurs once, the position map has the same size as the database.

In [107], Costa *et al.* also have explored the design of ORAM for SGX. They present a new design of hierarchical ORAM that is suitable for SGX. By using the multilevel adaptive hashing scheme [129], their design optimises the bandwidth cost and outperforms the Circuit ORAM [130] by $8\times$. However, this solution suffers from the same issue as [106], *i.e.,* it requires large long-term storage on SGX.

As summarised in Table 6.1, none of above schemes can achieve our objective, *i.e.,* preserving the access pattern privacy by using SGX while ensuring efficient search over encrypted data without any long-term storage on SGX.

6.8 Conclusions and Future Work

In this chapter, we introduce an approach that supports search over encrypted data and preserves the privacy of the access pattern using SGX. B+ tree indices are built in order to ensure

the search efficiency. By loading the tree nodes page by page and accessing the nodes in a balanced manner, the access pattern is also protected against page fault, timing, and cache attacks. Moreover, our solution can process large databases efficiently without requiring long-term storage on SGX.

In this chapter, we focus on the search operation and do not consider the insert and delete operations over the B+ tree. As future work, we aim to investigate how to securely support insert and delete queries. Moreover, we will explore the techniques to thwart other side channel attacks, such as the branch shadowing attack [111].

Chapter 7

Conclusions and Future Work

In this chapter, we summarise our contributions in this thesis and discuss possible future directions.

7.1 Summary

In cloud computing, SE could protect the outsourced data from powerful adversaries. However, state-of-the-art SE schemes are insecure due to sensitive information leakage, such as the search, access, and size patterns. In the literature, many recent works have shown that these patterns can be leveraged by adversaries to recover the content of queries and records. To effectively protect the outsourced data, it is necessary to minimise information leakage and ensure the forward and backward privacy simultaneously. In this thesis, we propose three different SE schemes for relational databases that not only ensure the confidentiality of the data, but also resist existing leakage-based attacks. Moreover, for the databases supporting multiple users to read and write the data, our SE schemes provide scalable key management methods that do not need to update the key or re-encrypting the data when revoking compromised users.

Before giving our solutions, we first identify the leakage issues remaining in existing SE schemes and define four different levels of leakage profiles for encrypted relational database solutions. After that, we investigate the existing leakage-based attacks. Based on the techniques and leakage leveraged on these attacks, we propose our countermeasures to thwart them. For instance, the access pattern can be protected by shuffling and re-randomising the searched records after executing each query. To protect the search and size patterns, it is necessary to randomise the encrypted queries and insert dummy records, respectively. Moreover, the correlation among the search, size, and access patterns should be broken. Otherwise, the attacker could infer the other two based on the leaked one. To break the correlation between the search and size patterns, our idea is to ensure all the queries either always match the same number of

records or always match different numbers of records even if the same query is repeated. We give solutions for both static and dynamic databases. The basic idea is to divide the elements in each field into groups and pad the elements in the same group into the same occurrence with dummy records. As a result, the queries searching for the terms in the same group will match the same number of records, and then the attackers cannot infer the search pattern of the queries in the same group based on their size patterns.

By combining the countermeasures, we propose three different solutions that minimise information leakage and withstand leakage-based attacks: *ObliviousDB*, *P-McDb*, and the SGX-assisted encrypted database. All of them can protect the search, access, and size patterns from the public CSPs effectively, and ensure the forward and backward privacy at the same time. *ObliviousDB*, *P-McDb*, and the SGX-assisted encrypted database are designed for the hybrid cloud, multi-cloud, and SGX-enabled cloud settings, respectively. Both *ObliviousDB* and *P-McDb* support multiple users to read and write the databases without sharing the secret keys among all the users. Furthermore, revoking a user does not require to updating the secrets keys and re-encrypting the records. The SGX-assisted encrypted database is a single-user scheme.

Comparing with *ObliviousDB*, *P-McDb* is significantly more efficient. Moreover, *P-McDb* does not rely on any trusted entities, but it suffers from the possible collusion attack between the two public CSPs. In addition, *P-McDb* requires more operations on the user side. Specifically, in *P-McDb*, the users have to filter out the returned dummy records before decryption, whereas, in *ObliviousDB*, this operation is performed by the OPS.

The role of WSS in *P-McDb* is similar to the role of the OPS in *ObliviousDB*. However, we cannot deploy the OPS in an untrusted public CSP, since it stores the flags that used to filter out dummy records for users in cleartext. On the contrary, the WSS could be deployed in a trusted private cloud. However, due to the storage of *NDB*, the WSS has a larger storage burden than the OPS.

Among the three proposed solutions, the SGX-assisted encrypted database is the most efficient one due to the usage of B+ tree indexing and the trusted code execution environment. Moreover, it supports complex queries and does not have collusion issues. However, it is currently a single-user scheme and only used for protecting static databases.

Above all, the three proposed schemes have both benefits and limitations. The cloud service customers can choose the scheme based on their resources and requirements.

7.2 Future Work

The research described in this thesis can be extended along several directions. In the following, we list several examples.

Leakage investigation for complex queries. In this thesis, the proposed solutions are mainly

designed to minimise information leakage for simple equality queries with only one predicate. When considering the queries with multiple predicates, our idea is to check if each predicate matches the record and determine the final result according to the conjunctions between the predicates. However, by doing so, the attacker might learn more information. One of our future directions is to analyse the information leaked when executing complex query apart from the search, access, and size patterns, and then investigate if an attacker could leverage the leakage to recover more useful information.

Active attacks detection. Another interesting direction is to ensure the correctness and integrity of search results against active malicious CSPs. In this thesis and the majority of other existing SE schemes, the CSP is assumed to be honest-but-curious, which means the CSP will follow the designed protocol honestly and return the correct result to users. However, the compromised CSP might not that honest and mount active attacks, such as tampering the data or returning random data to users. The techniques to check the correctness and integrity of the data are required in this scenario.

In addition, in both *ObliviousDB* and *P-McDb*, the access pattern privacy is protected by shuffling and re-randomising searched records. In *P-McDb*, the WSS is untrusted. It is necessary to ensure the WSS really performs the shuffling and re-randomising properly. Therefore, investigating the methods to supervise the behaviour of the untrusted CSPs is also one of our future work.

The application of SE schemes in emerging application scenarios. Initially, most of the SE schemes were designed to protect the file collections or relational databases stored in cloud servers. In recent years, SE schemes are gradually applied into several other scenarios, such as outsourced middleboxes [131–133], Content Delivery Network (CDN) [134–136] and publish/subscribe systems [137–139]. More recently, applications of SE schemes have received much attention from both the academia and the industry, such as the blockchain [140] and edge computing [141]. Using the concept of SE to protect the data in emerging application scenarios is also an interesting research direction.

References

[1] Amazon S3. https://aws.amazon.com/s3/. Last accessed: September 17, 2018. 1

[2] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004. 2

[3] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *S&P 2000*, pages 44–55. IEEE Computer Society, 2000. 2, 3, 62

[4] Muhammad Rizwan Asghar, Giovanni Russello, Bruno Crispo, and Mihaela Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *CCSW 2013*, pages 77–88. ACM, 2013. 2, 3, 5, 11, 14, 15, 19, 39, 42, 44, 64

[5] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):222–233, 2014. 2, 3, 11, 63

[6] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*. The Internet Society, 2014. 2, 3, 11

[7] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS 2005*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455, 2005. 2, 3, 11, 15, 23

[8] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS 2006*, pages 79–88. ACM, 2006. 2, 3, 11, 15, 63, 64

[9] Luca Ferretti, Fabio Pierazzi, Michele Colajanni, and Mirco Marchetti. Scalable architecture for multi-user encrypted SQL operations on cloud database services. *IEEE Trans. Cloud Computing*, 2(4):448–458, 2014. 2, 3, 11, 64

[10] Eu-Jin Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003. 2, 3

[11] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *SIGSAC 2014*, pages 310–320. ACM, 2014. 2, 3, 11, 13, 64

[12] Isabelle Hang, Florian Kerschbaum, and Ernesto Damiani. ENKI: access control for encrypted query processing. In *SIGMOD 2015*, pages 183–196. ACM, 2015. 2, 3, 11, 64

[13] Yong Ho Hwang and Pil Joong Lee. Public key encryption with conjunctive keyword search and its extension to a multi-user system. In *Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 2–22. Springer, 2007. 2, 3

[14] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS 2012*, pages 965–976. ACM, 2012. 2, 3, 11, 64

[15] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *FC 2013*, volume 7859 of *Lecture Notes in Computer Science*, pages 258–274. Springer, 2013. 2, 3, 11

[16] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic searchable encryption via blind storage. In *SP 2014*, pages 639–654. IEEE Computer Society, 2014. 2, 3, 11, 22, 62

[17] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP 2011*, pages 85–100. ACM, 2011. 2, 3, 11, 12, 14, 15, 39, 64

[18] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. DBMask: Fine-grained access control on encrypted relational databases. In *CODASPY 2015*, pages 1–11. ACM, 2015. 2, 3, 11, 12, 15, 64

[19] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2013*, volume 71, pages 72–75, 2014. 2, 3, 11, 23, 63

[20] Boyang Wang, Yantian Hou, Ming Li, Haitao Wang, and Hui Li. Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index. In *ASIA CCS 2014*, pages 111–122. ACM, 2014. 2, 3

[21] Bing Wang, Shucheng Yu, Wenjing Lou, and Y. Thomas Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *INFOCOM 2014*, pages 2112–2120. IEEE, 2014. 2, 3

[22] Yanjiang Yang, Joseph K. Liu, Kaitai Liang, Kim-Kwang Raymond Choo, and Jianying Zhou. Extended proxy-assisted approach: Achieving revocable fine-grained encryption of cloud data. In *ESORICS 2015*, volume 9327 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2015. 2, 3, 11, 64

[23] Attila Altay Yavuz and Jorge Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *SAC 2015*, volume 9566 of *Lecture Notes in Computer Science*, pages 241–259. Springer, 2015. 2, 3

[24] Wenhai Sun, Shucheng Yu, Wenjing Lou, Y. Thomas Hou, and Hui Li. Protecting your right: Attribute-based keyword search with fine-grained owner-enforced search authorization in the cloud. In *INFOCOM 2014*, pages 226–234. IEEE, 2014. 2, 3, 11, 64

[25] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, 2012. 2, 16

[26] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *SIGSAC 2015*, pages 644–655. ACM, 2015. 2, 15, 16

[27] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *SIGSAC 2015*, pages 668–679. ACM, 2015. 2, 12, 15, 17, 18, 20

[28] 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 Security 2016*, pages 707–720. USENIX Association, 2016. 2, 3, 15, 17

[29] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *SIGSAC 2016*, pages 1329–1340. ACM, 2016. 2, 13, 15, 18, 22

[30] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. *IACR Cryptology ePrint Archive*, 2017:24, 2017. 2, 17

[31] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373. Springer, 2013. 3, 11

[32] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *SIGSAC 2016*, pages 1143–1154. ACM, 2016. 3, 11, 23, 63

[33] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018. 3

[34] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS 2017*, pages 1465–1482. ACM, 2017. 3, 11, 23

[35] Cong Zuo, Shifeng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In *ESORICS 2018*, pages 228–246. Springer, 2018. 3, 11, 23

[36] Rafail Ostrovsky. Efficient computation on Oblivious RAMs. In *STOC 1990*, pages 514–523. ACM, 1990. 3

[37] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *SIGSAC 2013*, pages 299–310. ACM, 2013. 3, 13

[38] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996. 3, 13

[39] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999. 3, 22, 63

[40] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC 2009*, pages 169–178. ACM, 2009. 3

[41] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2016. 3, 6, 11, 13, 15, 18, 63, 85, 91

[42] Bharath Kumar Samanthula, Wei Jiang, and Elisa Bertino. Privacy-preserving complex query evaluation over semantically secure encrypted data. In *ESORICS 2014*, volume 8712 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2014. 3, 6, 11, 13, 15, 17, 22, 62, 70, 85

[43] Bing Wang, Wei Song, Wenjing Lou, and Y. Thomas Hou. Inverted index based multi-keyword public-key searchable encryption with strong privacy guarantee. In *INFOCOM 2015*, pages 2092–2100. IEEE, 2015. 3, 11, 22, 63

[44] RightScale 2018. https://www.rightscale.com/lp/state-of-the-cloud?campaign=7010g0000016JiA. Last accessed: August 7, 2018. 5, 6, 39

[45] Changyu Dong, Giovanni Russello, and Naranker Dulay. Shared and searchable encrypted data for untrusted servers. In *DBSec 2008*, volume 5094 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2008. 5, 39, 44

[46] Shujie Cui, Muhammad Rizwan Asghar, Steven D. Galbraith, and Giovanni Russello. Obliviousdb: Practical and efficient searchable encryption with controllable leakage. In *FPS 2017*, volume 10723 of *Lecture Notes in Computer Science*, pages 189–205. Springer, 2018. 5, 15

[47] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. Private database queries using somewhat homomorphic encryption. In *ACNS 2013*, pages 102–118. Springer, 2013. 6

[48] Adi Akavia, Dan Feldman, and Hayim Shaul. Secure search via multi-ring fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2018:245, 2018. 6

[49] Shujie Cui, Muhammad Rizwan Asghar, Steven D. Galbraith, and Giovanni Russello. P-McDb: Privacy-preserving search using multi-cloud encrypted databases. In *CLOUD 2017*, pages 334–341. IEEE Computer Society, 2017. 6, 15

[50] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016. 6, 93, 96, 113

[51] AMPLAB, University of Califorian. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/. Last accessed: March 2, 2018. 6, 95, 119

[52] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017. 6, 93, 94, 95, 119, 120

[53] Shujie Cui, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. Preserving access pattern privacy in sgx-assisted encrypted search. In *ICCCN 2018*. IEEE, 2018. 6, 15, 94

[54] Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *SIGSAC 2013*, pages 875–888. ACM, 2013. 11, 64

[55] Christoph Bösch, Andreas Peter, Bram Leenders, Hoon Wei Lim, Qiang Tang, Huaxiong Wang, Pieter H. Hartel, and Willem Jonker. Distributed searchable symmetric encryption. In *PST 2014*, pages 330–337. IEEE Computer Society, 2014. 11, 67, 91

[56] Guoxing Chen, Ten-Hwang Lai, Michael K. Reiter, and Yinqian Zhang. Differentially private access patterns for searchable symmetric encryption. In *INFOCOM 2018*, pages 810–818. IEEE, 2018. 11, 63

[57] Feng Bao, Robert H. Deng, Xuhua Ding, and Yanjiang Yang. Private query on encrypted data in multi-user settings. In *ISPEC 2008*, volume 4991 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2008. 11, 64

[58] Raluca A. Popa and Nickolai Zeldovich. Multi-key searchable encryption. *IACR Cryptology ePrint Archive*, 2013:508, 2013. 11, 64

[59] Qiang Tang. Nothing is for free: Security in searching shared and encrypted data. *IEEE Trans. Information Forensics and Security*, 9(11):1943–1952, 2014. 11, 64

[60] Aggelos Kiayias, Ozgur Oksuz, Alexander Russell, Qiang Tang, and Bing Wang. Efficient encrypted keyword search for multi-user data sharing. In *ESORICS 2016*, volume 9878 of *Lecture Notes in Computer Science*, pages 173–195. Springer, 2016. 11, 64

[61] Panagiotis Rizomiliotis and Stefanos Gritzalis. ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes. In *CCSW 2015*, pages 65–76. ACM, 2015. 11, 63

[62] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized I/O efficiency. *CoRR*, abs/1710.00183, 2017. 11, 23

[63] Shifeng 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*, pages 763–780. ACM, 2018. 11

[64] Thang Hoang, Attila Altay Yavuz, and Jorge Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *ACSAC 2016*, pages 302–313. ACM, 2016. 11, 67, 70, 91

[65] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. 12, 15

[66] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016. 13, 15, 16, 19, 63

[67] Ben A. Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M. Bellovin. Malicious-client security in blind seer: A scalable private DBMS. In *SP 2015*, pages 395–410. IEEE Computer Society, 2015. 13, 15

[68] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013. 15

[69] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *OSDI 2016*, pages 587–602. USENIX Association, 2016. 15

[70] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *SP 2014*, pages 359–374. IEEE Computer Society, 2014. 15, 17, 19

[71] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In *FC 2012*, volume 7397 of *Lecture Notes in Computer Science*, pages 285–298. Springer, 2012. 15

[72] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. Searchable encrypted relational databases: Risks and countermeasures. In *DPM 2017 and CBT 2017*, volume 10436 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2017. 15, 18

[73] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983. 16

[74] Enron email dataset. Last accessed: August 8, 2017. 16

[75] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998. 19

[76] Giovanni Di Crescenzo, Debra L. Cook, Allen McIntosh, and Euthimios Panagos. Practical private information retrieval from a time-varying, multi-attribute, and multiple-occurrence database. In *DBSec 2014*, volume 8566 of *Lecture Notes in Computer Science*, pages 339–355. Springer, 2014. 19

[77] Cédric Van Rompay, Refik Molva, and Melek Önen. Multi-user searchable encryption in the cloud. In *ISC 2015*, volume 9290 of *Lecture Notes in Computer Science*, pages 299–316. Springer, 2015. 19

[78] Jonathan Dautrich and Chinya V. Ravishankar. Combining ORAM with PIR to minimize bandwidth costs. In *CODASPY 2015*, pages 289–296. ACM, 2015. 19

[79] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO 2016*, volume 9816 of *Lecture Notes in Computer Science*, pages 563–592. Springer, 2016. 21

[80] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *TCC 2016-A*, volume 9563 of *Lecture Notes in Computer Science*, pages 145–174. Springer, 2016. 21

[81] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security 2015*, pages 415–430. USENIX Association, 2015. 21

[82] Muhammad Naveed. The fallacy of composition of oblivious RAM and searchable encryption. *IACR Cryptology ePrint Archive*, 2015:668, 2015. 21

[83] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptology ePrint Archive*, 2016:62, 2016. 23

[84] Libing Wu, Biwen Chen, Kim-Kwang Raymond Choo, and Debiao He. Efficient and secure searchable encryption protocol for cloud-based internet of things. *J. Parallel Distrib. Comput.*, 111:152–161, 2018. 23

[85] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS 2017*, pages 1465–1482. ACM, 2017. 23

[86] TPC-H. http://www.tpc.org/tpch/. Last accessed: August 15, 2017. 28, 58, 86, 115

[87] Raphael Bost and Pierre-Alain Fouque. Thwarting leakage abuse attacks against searchable encryption – a formal approach and applications to database padding. Cryptology ePrint Archive, Report 2017/1060, 2017. 28

[88] Muhammad Rizwan Asghar. *Privacy Preserving Enforcement of Sensitive Policies in Outsourced and Distributed Environments*. PhD thesis, University of Trento, Trento, Italy, December 2013. http://eprints-phd.biblio.unitn.it/1124/. 42

[89] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008. 44

[90] Muhammad Rizwan Asghar, Mihaela Ion, Giovanni Russello, and Bruno Crispo. $\text{Espoon}_{\text{erbac}}$: Enforcing security policies in outsourced environments. *Computers & Security*, 35:2–24, 2013. 48, 74

[91] Guomin Yang, Chik How Tan, Qiong Huang, and Duncan S. Wong. Probabilistic public key encryption with equality test. In *CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 119–131. Springer, 2010. 49

[92] Tianhao Wang and Yunlei Zhao. Secure dynamic SSE via access indistinguishable storage. In *AsiaCCS 2016*, pages 535–546. ACM, 2016. 62

[93] Konstantinos Chatzikokolakis, Miguel E. Andrés, Nicolás Emilio Bordenabe, and Catuscia Palamidessi. Broadening the scope of differential privacy using metrics. In *PETS 2013*, volume 7981 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2013. 63

[94] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *S&P 2007*, pages 321–334. IEEE Computer Society, 2007. 64

[95] Wenhai Sun, Xuefeng Liu, Wenjing Lou, Y. Thomas Hou, and Hui Li. Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data. In *INFOCOM 2015*, pages 2110–2118. IEEE, 2015. 64

[96] Rong Cheng, Jingbo Yan, Chaowen Guan, Fangguo Zhang, and Kui Ren. Verifiable searchable symmetric encryption from indistinguishability obfuscation. In *ASIACCS 2015*, pages 621–626. ACM, 2015. 64

[97] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *SIGSAC 2013*, pages 247–258. ACM, 2013. 67, 70, 90

[98] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. 77

[99] Ian F. Blake and Vladimir Kolesnikov. One-round secure comparison of integers. *J. Mathematical Cryptology*, pages 37–68, 2009. 85

[100] Cengiz Örencik, Ayse Selcuk, Erkay Savas, and Murat Kantarcioglu. Multi-keyword search over encrypted data with scoring and search pattern obfuscation. *Int. J. Inf. Sec.*, 15(3):251–269, 2016. 91

[101] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. Distributed search over encrypted big data. In *CODASPY 2015*, pages 271–278. ACM, 2015. 91

[102] Geong Sen Poh, Moesfa Soeheila Mohamad, and Ji-Jian Chin. Searchable symmetric encryption over multiple servers. *Cryptography and Communications*, 10(1):139–158, 2018. 91

[103] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and secure index with SGX. In *DBSec 2017*, pages 386–408. Springer, 2017. 93, 94, 98, 113, 116, 119

[104] A. Gribov, D. Vinayagamurthy, and S. Gorbunov. StealthDB: A Scalable Encrypted Database with Full SQL Query Support. *ArXiv e-prints*, November 2017. 94, 120

[105] 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 Aditya Akella and Jon Howell, editors, *NSDI 2017*, pages 283–298. USENIX Association, 2017. 93, 94, 119, 120

[106] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. *IACR Cryptology ePrint Archive*, 2017:549, 2017. 93, 94, 120

[107] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. The pyramid scheme: Oblivious RAM for trusted processors. *CoRR*, abs/1712.07882, 2017. 93, 94, 120

[108] Adil Ahmad, Kyungtae Kim, Ashish Kumar, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious file system for intel sgx. 2018. 94

[109] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SP 2015*, pages 640–656. IEEE Computer Society, 2015. 93, 94, 96, 98, 119

[110] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In William Enck and Collin Mulliner, editors, *WOOT 2017*. USENIX Association, 2017. 93, 94, 97

[111] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX 2017*, pages 557–574. USENIX Association, 2017. 93, 94, 98, 121

[112] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Securing data analytics on SGX with randomization. In *ESORICS 2017*, pages 352–369. Springer, 2017. 94

[113] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *AsiaCCS 2016*, pages 317–328. ACM, 2016. 94

[114] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017. 94

[115] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP 2013*, page 10, 2013. 96

[116] Introduction to Intel SGX sealing. `https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing`. Last accessed: March 4, 2018. 96

[117] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017. 97, 98

[118] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX 2003*. USENIX Association, 2003. 98

[119] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*, pages 1–20. Springer, 2006. 98

[120] J. Ambrose and A. Ignjatovic. *Power Analysis Side Channel Attacks: The Processor Design-level Context*. Omniscriptum Gmbh & Company Kg., 2010. 98

[121] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001. 98

[122] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000. 98

[123] Nested loop join. https://en.wikipedia.org/wiki/Nested_loop_join. Last accessed: April 8, 2018. 110

[124] Hash join. https://en.wikipedia.org/wiki/Hash_join. Last accessed: April 8, 2018. 110

[125] Sort-merge join. https://en.wikipedia.org/wiki/Sort-merge_join. Last accessed: April 8, 2018. 110

[126] GnuPG. https://www.gnupg.org/index.html. 115

[127] An oblivious general-purpose SQL database for the cloud. https://github.com/SabaEskandarian/ObliDB. Last accessed: March 4, 2018. 119

[128] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *EuroSys 2017*, pages 238–253. ACM, 2017. 120

[129] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *ACM-SIAM 1990*, pages 43–53. SIAM, 1990. 120

[130] Xiao Shaun Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *SIGSAC, 2015*, pages 850–861. ACM, 2015. 120

[131] Yu Guo, Cong Wang, and Xiaohua Jia. Enabling secure and dynamic deep packet inspection in outsourced middleboxes. In *AsiaCCS 2018*, pages 49–55. ACM, 2018. 125

[132] Xingliang Yuan, Huayi Duan, and Cong Wang. Assuring string pattern matching in outsourced middleboxes. *IEEE/ACM Trans. Netw.*, 26(3):1362–1375, 2018. 125

[133] Xingliang Yuan, Huayi Duan, and Cong Wang. Bringing execution assurances of pattern matching in outsourced middleboxes. In *ICNP 2016*, pages 1–10. IEEE Computer Society, 2016. 125

[134] Anne Edmundson, Paul Schmitt, Nick Feamster, and Jennifer Rexford. OCDN: oblivious content distribution networks. *CoRR*, abs/1711.01478, 2017. 125

[135] Jeremie Leguay, Georgios S. Paschos, Elizabeth A. Quaglia, and Ben Smyth. CryptoCache: Network caching with confidentiality. In *ICC 2017*, pages 1–6. IEEE, 2017. 125

[136] S. Cui, M. R. Asghar, and G. Russello. Multi-cdn: Towards privacy in content delivery networks. *IEEE Transactions on Dependable and Secure Computing*, 2018. 125

[137] Mihaela Ion, Giovanni Russello, and Bruno Crispo. Design and implementation of a confidentiality and access control solution for publish/subscribe systems. *Computer networks*, 56(7):2014–2037, 2012. 125

[138] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. Secure content-based routing using intel software guard extensions. In *Middleware 2016*, page 10. ACM, 2016. 125

[139] Shujie Cui, Sana Belguith, Pramodya De Alwis, Muhammad Rizwan Asghar, and Giovanni Russello. Malicious entities are in vain: Preserving privacy in publish and subscribe systems. In *The 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (Trustcom) 2018*, 2018. 125

[140] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data, BigData Congress 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 557–564. IEEE Computer Society, 2017. 125

[141] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata@MobiHoc 2015, Hangzhou, China, June 21, 2015*, pages 37–42. ACM, 2015. 125

List of My Publications

1. **Cui, Shujie**, Muhammad Rizwan Asghar, and Giovanni Russello. "Multi-CDN: Towards Privacy in Content Delivery Networks", IEEE Transactions on Dependable and Secure Computing (TDSC), **accepted**, 2018.

   In this work, we present a privacy-preserving encrypted Content Delivery Network (CDN) system to hide not only the content of objects and user requests, but also protect user preferences and the popularity of objects from curious CDN providers. We employ SSE to protect the objects and user requests in a way that both the CDNs and content providers can perform the search operations without accessing objects and requests in cleartext. Our proposed system is based on a scalable key management approach for multi-user access, where no key regeneration and data re-encryption are needed for user revocation.

2. **Cui, Shujie**, Sana Belguith, Pramodya De Alwis, Muhammad Rizwan Asghar, and Giovanni Russello. "Collusion Defender: Preserving Subscribers' Privacy in Publish and Subscribe Systems", IEEE Transactions on Dependable and Secure Computing (TDSC), **accepted**, 2019.

   In this article, we propose a solution that uses three different types of brokers and splits the matching operation into three phases, where each phase is executed by a different type of broker. Even in the case of malicious subscribers (or publishers) colluding with up two different types of brokers, they are unable to infer the subscriptions of innocent subscribers.

3. **Cui, Shujie**, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. "SGX-DB: Encrypted Database with Access Pattern Protection and Complex Query Support", IEEE Transactions on Dependable and Secure Computing (TDSC), **in submission**.

   The details of this paper are included in Chapter 6.

4. **Cui, Shujie**, Sana Belguith, Pramodya De Alwis, Muhammad Rizwan Asghar, and Giovanni Russello. "Malicious Entities are in Vain: Preserving Privacy in Publish and Sub-

scribe Systems", accepted in the 17th IEEE International Conference On Trust, Security and Privacy In Computing And Communications (TrustCom 2018).

This paper is a short version of the Collusion Defender.

5. **Cui, Shujie**, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. "Preserving Access Pattern Privacy in SGX-Assisted Encrypted Search", in the 27th International Conference on Computer Communications and Networks (IC3N 2018).

   The details of this paper are included in Chapter 6.

6. Belguith, Sana, **Shujie Cui**, Muhammad Rizwan Asghar, and Giovanni Russello. "Secure publish and subscribe systems with efficient revocation." In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (ACM SAC), pp. 388-394. ACM, 2018.

   In this paper, we propose a novel revocation technique for pub/sub systems that can efficiently remove compromised subscribers without requiring regeneration and redistribution of new keys as well as re-encryption of existing data with those keys. Our proposed solution is such that a subscriber's interest is not revealed to curious brokers and published data can only be accessed by the authorised subscribers. Finally, the proposed protocol is secure against the collusion attacks between brokers and revoked subscribers.

7. **Cui, Shujie**, Muhammad Rizwan Asghar, and Giovanni Russello. "Privacy-preserving content delivery networks." In Local Computer Networks (LCN), 2017 IEEE 42nd Conference on, pp. 607-610. IEEE, 2017.

   This paper is a short version of our Multi-CDN system.

8. **Cui, Shujie**, Muhammad Rizwan Asghar, Steven D. Galbraith, and Giovanni Russello. "P-McDb: Privacy-Preserving Search Using Multi-Cloud Encrypted Databases." In Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on, pp. 334-341. IEEE, 2017.

   The details of this paper are included in Chapter 5.

9. **Cui, Shujie**, Muhammad Rizwan Asghar, Steven D. Galbraith, and Giovanni Russello. "Secure and practical searchable encryption: a position paper." In Australasian Conference on Information Security and Privacy (ACISP), pp. 266-281. Springer, Cham, 2017.

   In this position paper, we briefly analyse the leakage-based attacks, and identify a set of requirements for a secure cloud database that could resist against these attacks and

ensure usability of the system simultaneously. We also discuss several possible solutions to fulfil the identified requirements.

10. **Cui, Shujie**, Muhammad Rizwan Asghar, Steven D. Galbraith, and Giovanni Russello. "ObliviousDB: Practical and Efficient Searchable Encryption with Controllable Leakage." In International Symposium on Foundations and Practice of Security (FPS), pp. 189-205. Springer, Cham, 2017.

    The details of this paper are included in Chapter 4.

11. **Cui, Shujie**, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. "Long White Cloud (LWC): A Practical and Privacy-Preserving Outsourced Database." In IFIP International Conference on Information Security Theory and Practice (WISTP), pp. 41-55. Springer, Cham, 2017.

    This paper investigates and proposes a practical hybrid cloud based privacy-preserving scheme, named *Long White Cloud* (LWC), for outsourced databases with a focus on providing security against statistical inferences. Performance is a key issue in the search and retrieval of encrypted databases. LWC supports logarithmic-time insert, search and delete queries executed by outsourced databases with minimised information leakage to curious CSPs.

12. **Cui, Shujie**, Muhammad Rizwan Asghar, and Giovanni Russello. "Towards Blockchain-based Scalable and Trustworthy File Sharing", in the 27th International Conference on Computer Communications and Networks (IC3N 2018).

    In this work, we propose a blockchain and proxy re-encryption based design for encrypted file sharing that brings a distributed access control and data management. By combining blockchain with proxy re-encryption, our approach not only ensures confidentiality and integrity of files, but also provides a scalable key management mechanism for file sharing among multiple users. Moreover, by storing encrypted files and related keys in a distributed way, our method can resist collusion attacks between revoked users and distributed proxies.