MONASH University

Enhanced Security for Searchable Symmetric Encryption Supporting Rich Queries

Cong Zuo

Doctor of Philosophy

A thesis submitted for the degree of *Doctor of Philosophy* at
Monash University in *2020*
Clayton School of Information Technology

Copyright notice

Abstract

Searchable symmetric encryption (SSE) allows a user to search on an encrypted database that is stored on the untrusted cloud while protecting the privacy of both the data and the queries. To enable updates on the encrypted database, dynamic SSE (DSSE) was proposed. It enables a client to perform updates and searches on encrypted data, which makes it very useful in practice. To protect DSSE from the leakage of updates (leading to break query or data privacy), two new security notions, forward and backward privacy, have been proposed recently. Although extensive attention has been paid to forward privacy, this is not the case for backward privacy. Backward privacy, first formally introduced by Bost et al., is classified into three types from weak to strong, exactly Type-III to Type-I. To the best of our knowledge, however, no practical DSSE schemes without trusted hardware (e.g., SGX) have been proposed so far, in terms of the strong backward privacy and constant roundtrips between the client and the server. Besides, the existing forward and backward private DSSE schemes either only support single keyword queries or require more interactions between the client and the server.

This research focuses on how to achieve forward and stronger backward private DSSE without trusted hardware and make it support rich queries. In this thesis, we propose some concrete forward/backward private DSSE schemes for range queries by using a binary tree data structure and give a new leakage function for range queries. Moreover, we present a new DSSE scheme by leveraging simple symmetric encryption with homomorphic addition and bitmap index. The new scheme can achieve both forward and backward privacy with one roundtrip. In particular, the backward privacy we achieve in our scheme (denoted by Type-I^{-}) is somewhat stronger than Type-I. In addition, our scheme is very practical as it involves only lightweight cryptographic operations. To make it scalable for supporting billions of files, we further extend it to a multi-block setting. Finally, we give the corresponding security proofs and experimental evaluation, which demonstrate both the security and practicality of our schemes, respectively.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Cong Zuo

Date: 18/09/2020

Publications during enrollment

Publications included in this thesis:

1. **Cong Zuo**, Shi-Feng Sun, Joseph K. Liu, Jun Shao, Josef Pieprzyk: Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward (and Backward) Security. In European Symposium on Research in Computer Security, pages 228-246, 2018. (Core Rank A)

2. **Cong Zuo**, Shi-Feng Sun, Joseph K. Liu, Jun Shao, Josef Pieprzyk: Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy. In European Symposium on Research in Computer Security, pages 283-303, 2019. (Core Rank A)

3. **Cong Zuo**, Shi-Feng Sun, Joseph K. Liu, Jun Shao, Josef Pieprzyk, Lei Xu: Forward and Backward Private DSSE for Range Queries. In IEEE Transactions on Dependable and Secure Computing, 2020. (Early Access, IF=6.404)

Other publications not included in this thesis:

1. Shi-Feng Sun, **Cong Zuo**, Joseph K. Liu, Amin Sakzad, Ron Steinfeld, Tsz Hon Yuen, Xingliang Yuan, Dawu Gu : Non-Interactive Multi-Client Searchable Encryption: Realization and Implementation. In IEEE Transactions on Dependable and Secure Computing, 2019. (Early Access, IF=6.404)

2. Shabnam Kasra Kermanshahi, Joseph K. Liu, Ron Steinfeld, Surya Nepal, Shangqi Lai, Randolph Loh, and **Cong Zuo**: Multi-client Cloud-based symmetric searchable encryption. In IEEE Transactions on Dependable and Secure Computing, 2019. (Early Access, IF=6.404)

3. Lei Xu, Shi-Feng Sun, Xingliang Yuan, Joseph K. Liu, **Cong Zuo**, Chungen Xu: Enabling Authorized Encrypted Search for Multi-Authority Medical Databases. In IEEE Transactions on Emerging Topics in Computing, 2019. (Early Access, IF=4.989)

4. Lei Xu, Chungen Xu, Joseph K. Liu, **Cong Zuo**, Peng Zhang: Building a Dynamic Searchable Encrypted Medical Database for Multi-client. In Information Sciences, 2019. (Early Access, IF=5.524)

5. Qingqing Gan, **Cong Zuo**, Jianfeng Wang, Shi-Feng Sun, Xiaoming Wang: Dynamic Searchable Symmetric Encryption with Forward and Backward Privacy: A Survey. In International Conference on Network and System Security, pages 37-52, 2019. (Core Rank B)

6. Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, **Cong Zuo**: Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In ACM SIGSAC Conference on Computer and Communications Security, pages 745-762, 2018. (Core Rank A*)

7. Randolph Loh, **Cong Zuo**, Joseph K. Liu, Shi-Feng Sun: A Multi-client DSSE Scheme Supporting Range Queries. In International Conference on Information Security and Cryptology, pages 289-307, 2018. (Core Rank B)

8. Zhimei Sui, Shangqi Lai, **Cong Zuo**, Xingliang Yuan, Joseph K. Liu, Haifeng Qian: An Encrypted Database with Enforced Access Control and Blockchain Validation. In International Conference on Information Security and Cryptology, pages 260-273, 2018. (Core Rank B)

9. Lei Xu, Chungen Xu, Joseph K. Liu, **Cong Zuo**, Peng Zhang: A Multi-client Dynamic Searchable Symmetric Encryption System with Physical Deletion. In International Conference on Information and Communications Security, pages 516-528, 2017. (Core Rank B)

10. Xinxin Ma, Jun Shao, **Cong Zuo**, Ru Meng: Efficient Certificate-Based Signature and Its Aggregation. In International Conference on Information Security Practice and Experience, pages 391-408, 2017. (Core Rank B)

Acknowledgements

Firstly, I would like to express my gratitude to my supervisors, Assoc. Prof. Joseph K. Liu, Dr. Shi-Feng Sun, Prof. Josef Pieprzyk, for their support and guidance. I learned a lot from them, including cryptographic knowledge and academic writing.

I also want to express my special thanks to Prof. Jun Shao, who led me to the palace of cryptography and recommended me to Joseph K. Liu to pursue a Ph.D. degree at Monash University. Moreover, he continued to give me help during the past few years. In addition, I want to thank Prof. Guiyi Wei and Prof. Yun Ling for helping me get the funding, which gives me financial support to visit Monash University before starting my Ph.D. journey. Without them, this would not have been possible.

My panel members Ron Steinfeld, Carsten Rudolph, and Xingliang Yuan, have given me their insightful comments about my research project, which help me to finish my research project and this thesis. So I would like to thank them for their valuable help.

Furthermore, I would like to acknowledge Data61, CSIRO, for the generous financial and resource support in the past three years.

I appreciate Ms. Danette Deriane for the help and the wish that she gave me when I was applying for the Ph.D. scholarship. Also, she continued to help me after I started my Ph.D. study. I also appreciate Ms. Kerry Mcmanus for the help that she gave me before and after my Ph.D. journey. Moreover, I want to thank Ms. Julie Holden for helping me with my milestones and improving my academic writing skills.

Finally, I am equally thankful to all the other teachers and staff for their generous assistance before and after my Ph.D. study.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Driven by the attractive on-demand features and advantages, the development and deployment of cloud-based applications have gained substantial attention in both the industry and research community. Cloud storage is one of the most successful cloud-based applications [1–4], since it provides users with efficient and effective storage services. However, once the user's data is uploaded to the cloud, the privacy of the data cannot be guaranteed since the cloud provider (or the hacker) could learn the data that the user stored on the cloud [5].

To protect the privacy of user's data, a naïve solution is to encrypt the data before outsourcing, which at the same time jeopardizes the usefulness of these data, such as searchability. To solve this dilemma between the usefulness and the confidentiality of the data, many research efforts have been proposed, e.g. Oblivious RAM (ORAM) [6, 7], Private Information Retrieval (PIR) [8, 9] and Searchable Symmetric Encryption (SSE) [10, 11]. Unfortunately, Oblivious RAM incurs large computation and client storage overhead, which makes it impractical in the large database, and Private Information Retrieval only protects the privacy of the queries, which allows a user to secretly retrieve the data without revealing the queries. Then searchable symmetric encryption (SSE), which enables searchability on the ciphertexts, is the most promising technique. Later, many works have been done in this area toward the scalability [12], the expressiveness [13, 14], the dynamism [15] or the verifiability of the searchable symmetric encryption.

Early SSE works in the static setting only, which means it does not support the update of the encrypted database. To circumvent this obstacle, dynamic searchable symmetric encryption (DSSE) has been proposed to enable updates on the encrypted database. It not only pertains the searchability over the encrypted data but also allows a user to update the encrypted database, which is very useful since the user's data is changing over time. However, during the update, it introduces more leakages that can be abused by the attackers [16]. For example, many DSSE schemes [15, 17] suffer from file injection attack [16, 18]. This attack can compromise the privacy of a client's query by injecting a small portion of new documents into the encrypted database.

To mitigate the above mentioned problem, two new security notions called forward and backward privacy were proposed. They were informally introduced by Stefanov et al. in 2014 [19]. Roughly speaking, for any adversary who may continuously observe the interactions between the server and the client, forward privacy is satisfied if the addition of

new files does not leak any information about previously queried keywords. In a similar vein, backward privacy holds if files that previously added and later deleted do not leak "too much" information within any period that two search queries on the same keyword happened[1]. Bost [20] formally defined forward privacy and designed a forward-private DSSE scheme, which is resistant against file-injection attacks [18]. Recently, Bost et al. [21] first gave formal backward privacy definitions with three different levels (Type-I to Type-III, from most secure to least secure) for dynamic searchable symmetric encryption scheme, and they also gave several concrete backward-private constructions with different privacy/efficiency trade-offs. Besides, a majority of published forward and backward private DSSE schemes support single keyword queries only. This greatly reduces their useability. In many applications, we need more expressive search queries, such as range queries, for instance.

Consider range queries. In a naïve solution, one could query all possible values in a range. This solution is not efficient if the range is large, as it requires a large communication overhead. To process range queries more efficiently and reduce communication cost, Faber et al. [13] applied a binary tree to the OXT scheme of Cash et al. [12]. Their solution works for static databases only and does support updates. To the best of our knowledge, none of the state-of-the-art searchable symmetric encryption schemes achieve all the aforementioned properties simultaneously.

1.2 Research Questions

To address the above problems, this research is going to solve the following question:

How to design the enhanced security for searchable symmetric encryption supporting rich queries?

In particular, we focuses on the following questions regarding the searchable symmetric encryption:

- **Forward/Backward private dynamic searchable symmetric encryption (DSSE) supporting rich queries.** Most existing dynamic searchable symmetric encryption schemes [15, 17] suffer from file-injection attacks. To mitigate these attacks, two new security notions, forward and backward privacy, have been proposed. However, most existing forward/backward private DSSE schemes only support single-keyword queries, which greatly impedes its prevalence in the real world. **Q1. How to design forward and/or backward private DSSE supporting more rich queries?**

[1]The files are leaked if the second search query is issued after the files are added but before they are deleted. This is unavoidable since the adversary can easily tell the difference of the search results before and after the same search query.

- **Efficient dynamic searchable symmetric encryption (DSSE) with forward and stronger backward privacy.** Recently, Bost et al. [21] introduced several DSSE schemes with different level of backward privacy. In particular, the scheme with Type-I backward privacy (most secure) is based on the ORAM technique, which is not efficient. **Q2. How to design efficient forward and stronger backward private DSSE (without using ORAM)?**

The above research questions comprise two requirements that need to be fulfilled in this research. The first one is that we need to design efficient forward/backward private DSSE supporting rich queries (e.g., range queries). The second one is to propose efficient DSSE with forward and stronger backward privacy.

1.3 Contributions

This thesis makes three major contributions. Specifically, it includes 2 original papers published in a conference and 1 original paper submitted in a peer-reviewed journal, and they are detailed in Chapter 3, 4 and 5, respectively. In particular, it presents efficient DSSE schemes supporting range queries with forward/backward privacy. Moreover, it introduces efficient DSSE schemes with forward and stronger backward privacy. Finally, it gives a more efficient forward and backward private DSSE for range queries. The following sections list all the DSSE schemes.

1.3.1 Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward/Backward Privacy

Chapter 3 introduces two efficient DSSE schemes supporting range queries with forward/backward privacy, which addresses the research question **Q1**.

The first DSSE scheme achieves forward privacy. Moreover, it supports range queries. To achieve this, we introduce a new binary tree and apply it to the framework of the scheme from [20]. The second one achieves backward privacy and supports range queries. To achieve backward privacy, we introduce the bit string representation and combine it with the Paillier cryptosystem [22]. See Chapter 3 for details.

1.3.2 Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy

To address research question **Q2**, Chapter 4 introduces two efficient DSSE schemes with forward and stronger backward privacy. Specifically, the backward privacy (named Type-I^{-}) we achieve is somewhat stronger than Type-I.

In Chapter 4, we first present the DSSE scheme with forward and stronger backward privacy. In this scheme, we use the bitmap index, where the addition and deletion of files can be achieved by module addition. To securely add the files, we introduce the simple symmetric encryption with homomorphic addition, which supports the addition of the encrypted files.

Theoretically, the bitmap can be of an arbitrary length, but a longer bitmap will significantly slow down modular operations. To address this problem, we divide the long bitmap into several shorter blocks. Then we extend the first scheme to the multi-block setting. See Chapter 4 for details.

1.3.3 Forward and Backward Private DSSE for Range Queries

The schemes in Chapter 4 supports single keyword queries only, which impedes its prevalence in practice. To make it support range queries, in Chapter 5, we refine the binary tree that is introduced in Chapter 3. Moreover, we introduce the backward privacy for our range queries named Type-R. Then we introduce a forward and Type-R backward private DSSE for range queries by applying the binary tree to the framework of the scheme from Chapter 4. See Chapter 5 for details.

1.4 Organization

The remaining chapters of this thesis are organized as follows. In Chapter 2, we give the related work for this thesis. In Chapter 3, we introduce the DSSE schemes supporting range queries with forward/backward privacy. Chapter 4 presents the DSSE schemes with forward and stronger backward privacy. Chapter 5 introduces the forward and backward private DSSE for range queries. Chapter 6 discuss some possible future directions. Finally, Chapter 7 concludes this thesis.

Chapter 2

Related Work

In this chapter, we will introduce the related works to this thesis. Note that, to keep consistency, we will introduce some of them in Chapter 3, 4 and 5, respectively.

2.1 Searchable Encryption

Generally speaking, there are two kinds of searchable encryption, one is searchable symmetric encryption, the other is public-key searchable encryption. Since the main focus of this research is on searchable symmetric encryption, we will give a detailed literature review about searchable symmetric encryption.

2.1.1 Searchable Symmetric Encryption

In 2000, Song et al. [10] were the first using symmetric key encryption to facilitate keyword search over the encrypted data. In their scheme, they encrypt each keyword in every document. For the search, they encrypt a keyword as the search token and send it to the server. Then the server compares each ciphertext in every document with the search token. In this case, the search time is linear with the number of keyword/file pairs, which is not efficient. To improve the efficiency, Goh [23] maps each keyword in a file to a bloom filter. Then the search time is linear with the number of files. Chang et al. [24] introduced a scheme with similar search complexity. To further improve the efficiency of SSE, Curtmola et al. [11] introduce the inverted index which can achieve sublinear search time. Moreover, they gave a formal definition for SSE and the corresponding security model in the static setting. Later, many SSE schemes with different properties have been introduced.

Searchable symmetric encryption supporting rich queries. Rich queries allow the client to query multiple keywords. In other words, the returned documents in the search results contain all these queried keywords, which can greatly facilitate the application of SSE in the real world. To achieve this, a naïve solution is to invoke single keyword queries for each queried keyword separately and calculate the intersections over the search results of each keyword. However, this method is not efficient, and it will introduce more leakages. In 2013, Cash et al. [12] introduced a sublinear SSE scheme (named Oblivious Cross-Tags (OXT)), which supports boolean queries. Their solution finds out a tradeoff between efficiency and security by carefully defining the leakage functions. Later, Zuo et al. [14] proposed a trusted boolean search on cloud using searchable symmetric encryption scheme which extends the scheme of Cash et al. [12] to support more general boolean formulas, while also significantly improving the efficiency over the scheme of Cash et al. [12] for many of the queries that

they do support. In 2015, Faber et al. [13] proposed a rich queries scheme beyond exact matches by extending the scheme of Cash et al. [12]. Lai et al. [25] constructed a new SSE protocol with support for conjunctive queries, named Hidden Cross-tags (HXT), enhancing the privacy-preserving while maintaining the search efficiency. By employing hidden vector encryption and bloom filter, their protocol prevents the keyword pair result pattern leakage during the conjunctive search procedure. Later, many SSE schemes with conjunctive queries were proposed, such as the schemes from [26–28].

Faber et al. [13] introduces an SSE scheme with range queries by extending the scheme from [12]. An SSE scheme with range queries can return all the documents or records within the requested range, which is quite useful in practical cloud storage. For example, a teacher wants to get all the students whose age is in the range (10,15). There is also another line of research to attack the range queries [29–31]. Note that, these attacks are based on the access pattern which is assumed to be leaked in most SSE schemes. In the future, we will investigate the SSE scheme without access pattern to mitigate these attacks.

Dynamic Searchable Symmetric Encryption. Dynamic searchable symmetric encryption (DSSE) has been proposed to protect the privacy of user's data that stored in the cloud (e.g. Google Drive [1]). It not only pertains the searchability over the encrypted data, but also allows the client to update the encrypted database after the setup which is very useful since user's data is changing over the time. However, many dynamic searchable symmetric encryption schemes [15, 17] suffer from file injection attack [16, 18]. This attack can compromise the privacy of a client's query by injecting a small portion of new documents to the encrypted database. To resist such an attack, Zhang et al. [18] highlighted the need of forward privacy which was first informally introduced by Stefanov et al. [19]. Forward privacy means an update does not leak the updated document matching a query we previously issued. In 2016, Bost [20] gave a formal forward privacy definition for dynamic searchable symmetric encryption schemes which slightly extended the informal definition of Stefanov et al. [19]. Bost also gave a concrete forward private dynamic searchable symmetric encryption scheme in [20]. In [19], Stefanov et al. also introduced the notion of backward privacy which means that a search query does not leak the file indices that previously added and later deleted. Until 2017, Bost et al. [21] first gave the formal backward privacy definition for dynamic searchable symmetric encryption scheme and they also gave several concrete backward private DSSE schemes. Nevertheless, the existing forward/backward private searchable symmetric encryption schemes only support single-keyword queries, which is not expressive enough in many application scenarios.

Security against malicious server model. Due to software/hardware failure or in order to save the computation and bandwidth, the server could return false results. Hence, there is a need of the verifiability over the results that returned from the server. To tackle this problem, Chai et al. [32] proposed a verifiable searchable encryption which provides the verifiability over the returned encrypted data. However, this scheme is not dynamic. Later, Bost et al. [33] proposed a verifiable dynamic searchable symmetric encryption which

offers not only the verifiability over the returned results from the server but also the forward privacy. They also gave the lower bounds on the computational complexity of search and update queries. The time complexity of efficient verifiable scheme is proportional to the number of operations corresponding to the search queries and the number of operations corresponding to the update queries. Nevertheless, this scheme only supports single keyword queries.

Efficiency and scalability of searchable symmetric encryption. Curtmola et al. [11] were the first using inverted index to achieve sub-linear search time. Some previous schemes, such as the seminar work of Song et al. [10], the search time was linear with the number of documents that stored on the cloud. However, these schemes only address the single keyword queries. Later, Cash et al. [12] proposed a highly scalable searchable symmetric encryption with boolean queries which is efficient. They also gave a detailed experimentation analysis on a real large dataset platforms. However, these schemes are not dynamic.

Multi-Client Searchable Symmetric Encryption. In the line of research of searchable symmetric encryption, the most basic setting is where the data owner is the one who performing the search on the encrypted database which is stored in a third party (e.g. the google drive [1]). The work of Curtmola et al. [11] was the first scheme to extend the two-party model (the data owner and the server) of searchable symmetric encryption to the multi-client setting. However, this scheme does not allow the interaction between the data owner and the client in each query which led to the inefficient implementation. To circumvent this obstacle, in 2010, Chase et al. [34] introduced a searchable symmetric encryption with controlled disclosure which allowed such interaction. Later, in 2011, to protect the privacy of a query, De Cristofaro et al. [35] proposed a privacy-preserving sharing of sensitive information scheme which extended the multi-client searchable symmetric encryption to the outsourced symmetric private information retrieval setting. Later, Jarecki et al. [36] introduced a more expressive and scalable searchable symmetric encryption with outsourced symmetric private information retrieval by using the technique of Cash et al. [12]. Sun et al. [37] proposed a non-interactive multi-client SSE scheme supporting boolean queries based on the framework of OXT [12]. Moreover, to avoid the interaction between the client and the server, they used the RSA function [38] to realize a non-interactive multi-client SSE scheme. Furthermore, the technique of attribute-based encryption [39–41] is embedded into their scheme to achieve fine-grained access control on the cloud data.

Chapter 3

Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward/Backward Privacy

Dynamic searchable symmetric encryption (DSSE) is a useful cryptographic tool in encrypted cloud storage. However, it has been reported that DSSE usually suffers from file-injection attacks and content leak of deleted documents. To mitigate these attacks, forward privacy and backward privacy have been proposed. Nevertheless, most existing forward/backward private DSSE schemes can only support single keyword queries.

In this chapter, we propose two DSSE schemes supporting range queries. One is forward-private and supports a large number of documents. The other can achieve backward privacy, while it can only support a limited number of documents. Finally, we also give the security proofs of the proposed DSSE schemes in the random oracle model.

3.1 Introduction

Searchable symmetric encryption (SSE) is a useful cryptographic primitive that can encrypt the data to protect its confidentiality while keeping its searchability. Dynamic SSE (DSSE) further provides data dynamics that allow the client to update data over time without losing data confidentiality and searchability. Due to this property, DSSE is highly demanded in an encrypted cloud. However, many existing DSSE schemes [15, 17] suffer from file-injection attacks [16, 18], where the adversary can compromise the privacy of a client query by injecting a small portion of new documents into the encrypted database. To resist this attack, Zhang et al. [18] highlighted the need for forward privacy that was informally introduced by Stefanov et al. [19]. The formal definition of forward privacy for DSSE was given by Bost [20], who also proposed a concrete forward-private DSSE scheme. Furthermore, Bost et al. [21] demonstrated the damage of content leak of deleted documents and proposed the corresponding security notion—backward privacy. Several backward-private DSSE schemes were also presented in [21].

Nevertheless, the existing forward/backward private DSSE schemes only support single keyword queries, which are not expressive enough in data search service [13, 42]. To solve this problem, we aim to design forward/backward private DSSE schemes supporting range queries. Our design starts from the regular binary tree in [13] to support range queries. However, the binary tree in [13] cannot be applied directly to the dynamic setting. It is mainly because that the keywords in [13] are labeled according to the corresponding tree levels that will change significantly in the dynamic setting. A naïve solution is to replace all

old keywords with the associated new keywords. This is, however, not efficient. To address this problem, we have to explore new approaches to our goal.

Our Contributions. To achieve the above goal, we propose two new DSSE constructions supporting range queries in this chapter. The first one is forward private but with a larger client overhead in contrast to [13]. The second one is backward private DSSE, which greatly reduces the client and the server storage at the cost of losing forward privacy. In more details, our main contributions are as follows:

- To make the binary tree suitable for range queries in the dynamic setting, we introduce a new binary tree data structure, and then present the first forward private DSSE supporting range queries by applying it to Bost's scheme [20]. However, forward privacy is achieved at the expense of suffering from a large storage overhead on the client side. In particular, client storage is two times larger than the client storage of Bost's scheme.
- To achieve backward privacy, we apply the Paillier cryptosystem and the bit string representation to Bost's framework. To reduce the storage at both the client and server side, we used a fixed update token for each keyword. Note that this scheme is not forward private since, for every update, the server knows which keyword has been updated. We refer readers to Section 3.4 for more details. Notably, due to the limitation of the Paillier cryptosystem, it cannot support a large-scale database consisting of a large number of documents. Nevertheless, it suits well for certain scenarios where the number of documents is moderate. The new approach may give new lights on designing more efficient and secure DSSE schemes.
- Also, the comparison with related works in Table 3.1 and detailed security analyses are provided, which demonstrate that our constructions are not only forward/backward private but also with comparable efficiency.

Table 3.1: Comparison with existing DSSE schemes

Scheme	Client Computation		Client Storage	Range Queries	Forward Privacy	Backward Privacy	Document number
	Search	Update					
[13]	w_R	-	$O(1)$	✓	✗	✗	large
[20]	-	$O(1)$	$O(W)$	✗	✓	✗	large
Ours A	w_R	$\lceil log(W) \rceil + 1$	$O(W)$	✓	✓	✗	large
Ours B	w_R	$\lceil log(W) \rceil + 1$	$O(1)$	✓	✗	✓	small

W is the number of keywords in a database. w_R is the number of keywords for a range query(we map a range query to a few different keywords).

Remark: In this chapter, we correct the wrong theorem (cf. Theorem 2) in the conference version [43], where we claimed that the second construction could achieve forward privacy. In fact, forward privacy cannot be achieved because the update can be linked to previous

searches by the fixed token. In particular, we focus on achieving backward privacy by applying the Paillier cryptosystem and the bit string representation to Bost [20]'s framework, which can also achieve forward privacy due to Bost's technique. For every update on keyword n, however, the server needs to store a ciphertext (of Paillier encryption) corresponding to n. In addition, the number of keywords is nearly doubled compared to the scheme of [20], as mentioned in our first construction. Then this construction incurs large storage on both the server and the client side. In the conference version, we used a fixed update token for each keyword to further reduce the storage overhead. In this way, the server can homomorphically add the ciphertext to the previous ones corresponding to the same keyword, and the client does not need to store the current search token yet. Therefore, both the client and the server storage are reduced a lot, but at the cost of losing forward privacy. Nevertheless, we made some mistakes when preparing the conference version and so we correct it here.

3.1.1 Related Work

For the completeness and consistency of this Chapter, we list some related works that already appeared in Section 2.1.1. Song et al. [10] were the first using symmetric encryption to facilitate keyword search over the encrypted data. Later, Curtmola et al. [11] gave a formal definition for SSE and the corresponding security model in the static setting. To make SSE more scalable and expressive, Cash et al. [12] proposed a new scalable SSE supporting Boolean queries. Following this construction, many extensions have been proposed. Faber et al. [13] extended it to process a much richer collection of queries. For instance, they used a binary tree with keywords labeled according to the tree levels to support range queries. Zuo et al. [14] made another extension to support general Boolean queries. Cash et al.'s construction has also been extended into a multi-user setting [37, 44, 45]. However, the above schemes cannot support data updates. To solve this problem, some DSSE schemes have been proposed [15, 17].

However, designing a secure DSSE scheme is not an easy job. Cash et al. [16] pointed out that only a small leakage leveraged by the adversary would be enough to compromise the privacy of clients' queries. A concrete attack named file-injection attack was proposed by Zhang et al. [18]. In this attack, the adversary can infer the concept of client queries by injecting a small portion of new documents into an encrypted database. This attack also highlights the need for forward privacy, which protects the security of new added parts. Accordingly, we have backward privacy that protects the security of new added parts and later deleted. These two security notions were first introduced by Stefanov et al. [19]. The formal definitions of forward/backward privacy for DSSE were given by Bost [20] and Bost et al. [21], respectively. In [20], Bost also proposed a concrete forward private DSSE scheme; it does not support physical deletion. Later on, Kim et al. [46] proposed a forward private DSSE scheme supporting physical deletion. Meanwhile, Bost et al. [21] proposed a

forward/backward-private DSSE to reduce leakage during deletion. Unfortunately, all the existing forward/backward-private DSSE schemes only support single keyword queries. Hence, forward/backward-private DSSE supporting more expressive queries, such as range queries, are quite desired.

Apart from the binary tree technique, order preserving encryption (OPE) can also be used to support range queries. The concept of OPE was proposed by Agrawal et al. [47], and it allows the order of the plaintexts to be preserved in the ciphertexts. It is easy to see that this kind of encryption would lead to leakage in [48, 49]. To reduce this leakage, Boneh et al. [50] proposed another concept named order revealing encryption (ORE), where the order of the ciphertexts are revealed by using an algorithm rather than comparing the ciphertexts (in OPE) directly. More efficient ORE schemes were proposed later [51]. However, ORE-based SSE still leaks much information about the underlying plaintexts. To avoid this, in this Chapter, we focus on how to use the binary tree structure to achieve range queries.

3.1.2 Organization

The remaining sections of this Chapter are organized as follows. In Sect. 3.2, we give the background information and building blocks that are used in this Chapter. In Sect. 3.3, we give the definition of DSSE and its security definition. After that in Sect. 3.4, we present a new binary tree and our DSSE schemes. Their security analyses are given in Sect. 3.5. Finally, Sect. 3.6 concludes this chapter.

3.2 Preliminaries

In this section, we describe cryptographic primitives (building blocks) that are used in this chapter.

3.2.1 Trapdoor Permutations

A trapdoor permutation (TDP) Π is a one-way permutation over a domain D such that (1) it is "easy" to compute Π for any value of the domain with the public key, and (2) it is "easy" to calculate the inverse Π^{-1} for any value of a co-domain $\mathcal{M}$ only if a matching secret key is known. More formally, Π consists of the following algorithms:

- $\mathtt{TKeyGen}(1^\lambda) \rightarrow (\mathtt{TPK}, \mathtt{TSK})$: For a security parameter 1^λ, the algorithm returns a pair of cryptographic keys: a public key $\mathtt{TPK}$ and a secret key $\mathtt{TSK}$.
- $\Pi(\mathtt{TPK}, x) \rightarrow y$: For a pair: public key $\mathtt{TPK}$ and $x \in D$, the algorithm outputs $y \in \mathcal{M}$.
- $\Pi^{-1}(\mathtt{TSK}, y) \rightarrow x$: For a pair: a secret key $\mathtt{TSK}$ and $y \in \mathcal{M}$, the algorithm returns $x \in D$.

One-wayness. We say Π is one-way if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, an advantage

$$\mathtt{Adv}^{\mathtt{OW}}_{\Pi,\mathcal{A}}(1^\lambda) = \Pr[x \leftarrow \mathcal{A}(\mathtt{TPK}, y)]$$

is negligible, where $(\mathtt{TSK}, \mathtt{TPK}) \leftarrow \mathtt{TKeyGen}(1^\lambda)$, $y \leftarrow \Pi(\mathtt{TPK}, x)$, $x \in D$.

3.2.2 Paillier Cryptosystem

A Paillier cryptosystem $\Sigma = (\mathtt{KeyGen}, \mathtt{Enc}, \mathtt{Dec})$ is defined by following three algorithms:

- $\mathtt{KeyGen}(1^\lambda) \rightarrow (\mathtt{PK}, \mathtt{SK})$: It chooses at random two primes p and q of similar lengths and computes $n = pq$ and $\phi(n) = (p-1)(q-1)$. Next it sets $g = n+1$, $\beta = \phi(n)$ and $\mu = \phi(n)^{-1} \bmod n$. It returns $\mathtt{PK} = (n, g)$ and $\mathtt{SK} = (\beta, \mu)$.
- $\mathtt{Enc}(\mathtt{PK}, m) \rightarrow c$: Let m be the message, where $0 \leq m < n$, the algorithm selects an integer r at random from $\mathbb{Z}_n$ and computes a ciphertext $c = g^m \cdot r^n \bmod n^2$.
- $\mathtt{Dec}(\mathtt{SK}, c) \rightarrow m$: The algorithm calculates $m = L(c^\beta \bmod n^2) \cdot \mu \bmod n$, where $L(x) = \frac{x-1}{n}$.

Semantic Security. We say Σ is semantically secure if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, an advantage

$$\mathtt{Adv}^{\mathtt{IND-CPA}}_{\Sigma,\mathcal{A}}(1^\lambda) = |\Pr[\mathcal{A}(\mathtt{Enc}(\mathtt{PK}, m_0)) = 1] - \Pr[\mathcal{A}(\mathtt{Enc}(\mathtt{PK}, m_1)) = 1]|$$

is negligible, where $(\mathtt{SK}, \mathtt{PK}) \leftarrow \mathtt{KeyGen}(1^\lambda)$, $\mathcal{A}$ chooses m_0, m_1 and $|m_0| = |m_1|$.

Homomorphic Addition. Paillier cryptosystem is homomorphic, i.e.

$$\mathtt{Dec}(\mathtt{Enc}(m_1) \cdot \mathtt{Enc}(m_2)) \bmod n^2 = m_1 + m_2 \bmod n.$$

In our second construction, we need this property to achieve backward privacy.

3.2.3 Notations

The list of notations used is given in Table 3.2.

3.3 Dynamic Searchable Symmetric Encryption (DSSE)

We follow the database model given in the paper [20]. A database is a collection of (index, keyword set) pairs denoted as $\mathtt{DB} = (ind_i, \mathbf{W}_i)_{i=1}^d$, where $ind_i \in \{0,1\}^\ell$ and $\mathbf{W}_i \subseteq \{0,1\}^*$. The set of all keywords of the database $\mathtt{DB}$ is $\mathbf{W} = \cup_{i=1}^d \mathbf{W}_i$, where d is the number of documents in $\mathtt{DB}$. We identify $W = |\mathbf{W}|$ as the total number of keywords and $N = \Sigma_{i=1}^d |\mathbf{W}_i|$

Table 3.2: Notations (used in our constructions)

W	The number of keywords in a database DB
BDB	The binary database which is constructed from a database DB by using our binary tree BT
m	The number of values in the range $[0, m-1]$ for our range queries
v	A value in the range $[0, m-1]$ where $0 \leq v < m$
n_i	The i-th node in our binary tree which is considered as the keyword
$\texttt{root}_o$	The root node of the binary tree before update
$\texttt{root}_n$	The root node of the binary tree after update
ST_c	The current search token for a node n
$\mathcal{M}$	A random value for ST_0 which is the first search token for a node n
UT_c	The current update token for a node n
T	A map which is used to store the encrypted database EDB
N	A map which is used to store the current search token for n_i
NSet	The node set which contains the nodes
TPK	The public key of trapdoor permutation
TSK	The secret key of trapdoor permutation
PK	The public key of Paillier cryptosystem
SK	The secret key of Paillier cryptosystem
f_i	The i-th file
PBT	Perfect binary tree
CBT	Complete binary tree
VBT	Virtual perfect binary tree
ABT	Assigned complete binary tree

as the number of document/keyword pairs. We denote DB(w) as the set of documents that contain a keyword w. To achieve a sublinear search time, we encrypt the file indices of DB(w) corresponding to the same keyword w (a.k.a. inverted index[1]).

A DSSE scheme Γ consists of an algorithm **Setup** and two protocols **Search** and **Update** as described below.

- (EDB, σ) $\leftarrow$ **Setup**(DB, 1^λ): For a security parameter 1^λ and a database DB. The algorithm outputs an encrypted database EDB for the server and a secret state σ for the client.
- ($\mathcal{I}$, $\perp$) $\leftarrow$ **Search**(q, σ, EDB): The protocol is executed between a client (with her query q and state σ) and a server (with its EDB). At the end of the protocol, the client outputs a set of file indices $\mathcal{I}$ and the server outputs nothing.
- (σ', EDB$'$) $\leftarrow$ **Update**(σ, op, in, EDB): The protocol runs between a client and a server. The client input is a state σ, an operation $op = (add, del)$ she wants to perform and a collection of $in = (ind, \mathbf{w})$ pairs that are going to be modified, where add, del mean the

[1]It is an index data structure where a word is mapped to a set of documents which contain this word.

addition and deletion of a document/keyword pair, respectively, ind is the file index and **w** is a set of keywords. The server input is $\mathtt{EDB}$. **Update** returns an updated state σ' to the client and an updated encrypted database $\mathtt{EDB}'$ to the server.

3.3.1 Security Definition

The security definition of DSSE is formulated using the following two games: $\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda})$ and $\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda})$. The $\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda})$ is executed using DSSE. The $\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda})$ is simulated using the leakage of DSSE. The leakage is parameterized by a function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$, which describes what information is leaked to the adversary $\mathcal{A}$. If the adversary $\mathcal{A}$ cannot distinguish these two games, then we can say there is no other information leaked except the information that can be inferred from the leakage function $\mathcal{L}$. More formally,

- $\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda})$: On input a database $\mathtt{DB}$, which is chosen by the adversary $\mathcal{A}$, it outputs $\mathtt{EDB}$ by using **Setup**$(1^{\lambda}, \mathtt{DB})$ to the adversary $\mathcal{A}$. $\mathcal{A}$ can repeatedly perform a search query q (or an update query (op, in)). The game outputs the results generated by running **Search**(q) (or **Update**(op, in)) to the adversary $\mathcal{A}$. Eventually, $\mathcal{A}$ outputs a bit.

- $\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda})$: On input a database $\mathtt{DB}$ which is chosen by the adversary $\mathcal{A}$, it outputs $\mathtt{EDB}$ to the adversary $\mathcal{A}$ by using a simulator $\mathcal{S}(\mathcal{L}^{Stp}(1^{\lambda}, \mathtt{DB}))$. Then, it simulates the results for the search query q by using the leakage function $\mathcal{S}(\mathcal{L}^{Srch}(q))$ and uses $\mathcal{S}(\mathcal{L}^{Updt}(op, in))$ to simulate the results for update query (op, in). Eventually, $\mathcal{A}$ outputs a bit.

Definition 1. *A DSSE scheme Γ is $\mathcal{L}$-adaptively-secure if for every PPT adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ such that*

$$|\Pr[\mathtt{DSSEREAL}^{\Gamma}_{\mathcal{A}}(1^{\lambda}) = 1] - \Pr[\mathtt{DSSEIDEAL}^{\Gamma}_{\mathcal{A},\mathcal{S}}(1^{\lambda}) = 1]| \leq negl(1^{\lambda}).$$

3.4 Constructions

In this section, we give two DSSE constructions. In order to process range queries, we deploy a new binary tree, which is modified from the binary tree in [13]. Now, we first give our binary tree used in our constructions.

3.4.1 Binary Tree for Range Queries

In a binary tree $\mathtt{BT}$, every node has at most two children named $left$ and $right$. If a node has a child, then there is an edge that connects these two nodes. The node is the parent

$parent$ of its child. The root $root$ of a binary tree does not have parent and the leaf of a binary tree does not have any child. In this chapter, the binary tree is stored in thew form of linked structures. The first node of BT is the root of a binary tree. For example, the root node of the binary tree BT is BT, the left child of BT is BT.$left$, and the parent of BT's left child is BT.$left.parent$, where BT = BT.$left.parent$.

In a complete binary tree CBT, every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible (the leaf level may not full). A perfect binary tree PBT is a binary tree in which all internal nodes (not the leaves) have two children, and all leaves have the same depth or same level. Note that, PBT is a special CBT.

3.4.2 Binary Database

In this chapter, we use binary database BDB which is generated from DB. In DB, keywords (the first row in 3.1.(c)) are used to retrieve the file indices (every column in 3.1.(c)). For simplicity, we map keywords in DB to the values in the range [0,$m-1$] for range queries[2], where m is the maximum number of values. If we want to search the range [0,3], a naïve solution is to send every value in the range (0, 1, 2, and 3) to the server, which is not efficient. To reduce the number of keywords sent to the server, we use the binary tree as shown in Fig. 3.1.(a). For the range query [0,3], we simply send the keyword n_3 (the minimum nodes to cover value 0, 1, 2, and 3) to the server. In BDB, every node in the binary tree is the keyword of the binary database, and every node has all the file indices for its decedents, as illustrated in Figure 3.1.(d).

As shown in Fig. 3.1.(a), keyword in BDB corresponding to node i (the black integer) is n_i (e.g. the keyword for node 0 is n_0.). The blue integers are the keywords in DB and are mapped to the values in the range [0,3]. These values are associated with the leaves of our binary tree. The words in red are the file indices in DB. For every node (keyword), it contains all the file indices in its descendant leaves. Node n_1 contains f_0, f_1, f_2, f_3 and there is no file in node n_4 (See Fig. 3.1.(d)). For a range query $[0, 2]$, we need to send the keywords n_1, n_4 (n_1 and n_4 are the minimum number of keywords to cover the range $[0, 2]$.) to the server, and the result file indices are f_0, f_1, f_2 and f_3.

Bit String Representation.

We parse the file indices for every keyword in BDB (e.g. every column in Figure 3.1.(d)) into a bit string, which we will use later. Suppose there are $y-1$ documents in our BDB, then we need y bits to represent the existence of these documents. The highest bit is the sign bit (0 means positive and 1 means negative). If f_i contains keyword n_j, then the i-th bit of the bit string for n_j (every keyword has a bit string) is set to 1. Otherwise, it is set to 0. For the update, if we want to add a new file index f_i (which also contains keyword n_j) to

[2]In different applications, we can choose different kinds of values. For instance, audit documents of websites with particular IP addresses. We can search the whole network domain, particular host, or application range.

keyword n_j, we need a positive bit string, where the i-th bit is set to 1, and all other bits are set to 0. Next, we add this bit string to the existing bit string associated with n_j [3]. Then, f_i is added to the bit string for n_j. If we want to delete file index f_i from the bit string for n_j, we need a negative bit string (the most significant bit is set to 1), the i-th bit is set to 1, and the remaining bits are set to 0. Then, we need to get the complement of the bit string [4]. Next, we add the complement bit string as in the add operation. Finally, the f_i is deleted from the bit string for n_j.

For example, in Fig. 3.1.(b), the bit string for n_0 is 000001, and the bit string for n_4 is 000000. Assume that we want to delete file index f_0 from n_0 and add it to n_4. First, we need to generate bit string 000001 and add it to the bit string (000000) for n_4. Next, we generate the complement bit string 111111 (the complement of 100001) and add it to 000001 for n_0. Then, the result bit strings for n_0 and n_4 are 000000 and 000001, respectively. As a result, the file index f_0 has been moved from n_0 to n_4.

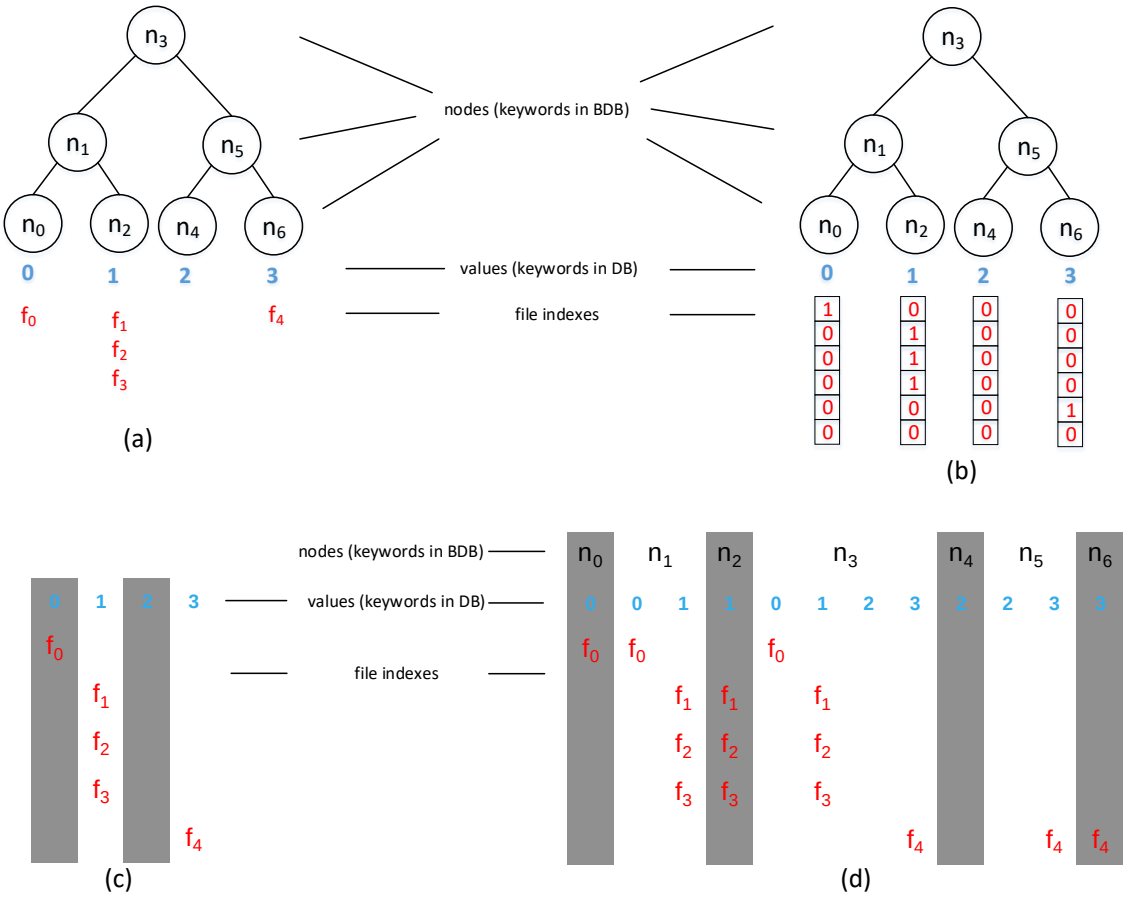


Figure 3.1: Architecture of Our Binary Tree for Range Queries

Binary Tree Assignment and Update.

As we use the binary tree to support the data structure needed in our DSSE, we define the following operations that are necessary to manipulate the DSSE data structure.

[3]Note that, in the range queries, the bit strings are bit exclusive since a file corresponds to one value only.
[4]In a computer, and the subtraction is achieved by adding the complement of the negative bit string.

TCon(m): For an integer m, the operation builds a complete binary tree CBT. CBT has $\lceil log(m) \rceil + 1$ levels, where the root is on the level 0, and the leaves are on the level $\lceil log(m) \rceil$. All leaves are associated with the m consecutive integers from left to right.

TAssign(CBT): The operation takes a CBT as an input and outputs an assigned binary tree ABT, where nodes are labelled by appropriate integers. The operation applies **TAssignSub** recursively. Keywords then are assigned to the node integers.

TAssignSub(c, CBT): For an input pair: a counter c and CBT, the operation outputs an assigned binary tree. It is implemented as a recursive function. It starts from 0 and assigns to nodes incrementally. See Fig. 3.2 for an example.

Algorithm 1 Our Binary Tree

TCon(m)
Input integer m
Output complete binary tree CBT
1: Construct a CBT with $\lceil log(m) \rceil + 1$ levels.
2: Set the number of leaves to m.
3: Associate the leaves with m consecutive integers [0,m-1] from left to right.
4: **return** CBT

TAssign(CBT)
Input complete binary tree CBT
Output assigned binary tree ABT
1: Counter $c = 0$
2: **TAssignSub**(c, CBT)
3: **return** ABT

TAssignSub(c, CBT)
Input CBT, counter c
Output Assigned binary tree ABT
1: **if** CBT.$left \neq \perp$ **then**
2: **TAssignSub**(c, CBT.$left$)
3: **end if**
4: Assign CBT with counter c.
5: $c = c + 1$
6: **if** CBT.$right \neq \perp$ **then**
7: **TAssignSub**(c, CBT.$right$)
8: **end if**
9: Assign CBT with counter c.
10: $c = c + 1$
11: **return** ABT

TGetNodes(n, ABT)
Input node n, ABT
Output NSet
1: **NSet** $\leftarrow$ Empty Set
2: **while** $n \neq \perp$ **do**
3: **NSet** $\leftarrow$ **NSet** $\cup\, n$
4: $n = n.parent$
5: **end while**
6: **return NSet**

TUpdate(add, v, CBT)
Input op= add, value v, CBT
Output updated CBT
1: **if** CBT $= \perp$ **then**
2: Create a node.
3: Associate value $v = 0$ to this node.
4: Set CBT to this node.
5: **else if** CBT is PBT or CBT has one node **then**
6: Create a new root node $\texttt{root}_n$.
7: Create a VBT = CBT
8: CBT.$parent$=VBT.$parent$=$\texttt{root}_n$
9: CBT = $\texttt{root}_n$
10: Associate v to the least virtual leaf and set this leaf and its parents as real.
11: **else**
12: Execute line 10.
13: **end if**
14: **return** CBT

TGetNodes(n, ABT): For an input pair: a node n and a tree ABT, the operation generates a collection of nodes in a path from the node n to the root node. This operation is needed for our update algorithm if a client wants to add a file to a leaf (a value in the range). The file is added to the leaf and its parent nodes.

TUpdate(add, v, CBT): The operation takes a value v and a complete binary tree CBT and updates CBT so the tree contains the value v. For simplicity, we consider the current complete binary tree contains values in the range $[0, v-1]$[5]. Depending on the value of v, the operation is executed according to the following cases:

- $v = 0$: It means that the current complete binary tree is null, we simply create a node and associate value $v = 0$ with the node. The operation returns the node as CBT.
- $v > 0$: If the current complete binary tree is a perfect binary tree PBT or it consists of a single node only, we need to create a virtual binary tree VBT, which is a copy of the current binary tree. Next, we merge the virtual perfect binary tree with the original one getting a large perfect binary tree. Finally, we need to associate the value v with the least virtual leaf (the leftmost virtual leaf without a value) of the virtual binary tree and set this leaf and its parents as real. For example, in Fig. 3.2.(a), $v = 4$, the nodes with the solid line are real, and the nodes with the dotted line are virtual, which can be added later. Otherwise, we directly associate the value v to the least virtual leaf and set this leaf and its parents as real [6]. In Fig. 3.2.(b), $v = 5$.

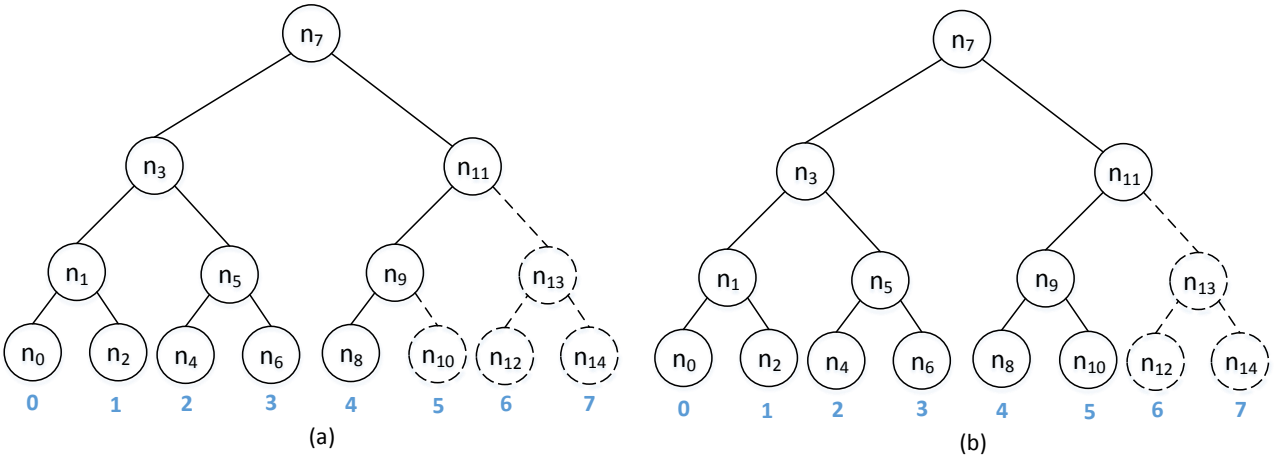


Figure 3.2: Example of Update Operation

Note that, in our range queries, we need to parse a normal database DB to its binary form BDB. First, we need to map keywords of DB to integers in the range $[0, |W|-1]$, where $|W|$ is the total number of keywords in DB. Next, we construct a binary tree as described above. The keywords are assigned to the nodes of the binary tree and are associated with the documents of their descendants. For example, In Fig. 3.1.(a), the keywords are $\{n_0, n_1, \cdots, n_6\}$ and $\mathtt{BDB}(n_0) = \{f_0\}$, $\mathtt{BDB}(n_1) = \{f_0, f_1, f_2, f_3\}$.

[5]Note that, we can use **TUpdate** many times if we need to update more values.
[6]Only if its parents were virtual, then we need to convert them to real.

3.4.3 DSSE Range Queries - Construction A

In this section, we apply our new binary tree to the Bost [20] scheme to support range queries. For performing a ranger query, the client in our scheme first determines a collection of keywords to cover the requested range. Then, she generates the search token corresponding to each node (in the cover) and sends them to the server, which can be done in a similar way as [20]. Now we are ready to present the first DSSE scheme that supports range queries and is forward private. The scheme is described in Algorithm 2, where F is a cryptographically strong pseudorandom function (PRF), H_1 and H_2 are keyed hash functions and Π is a trapdoor permutation.

Setup(1^λ): For a security parameter 1^λ , the algorithm outputs ($\texttt{TPK}, \texttt{TSK}, K, \mathbf{T}, \mathbf{N}, m$), where TPK and TSK are the public key and secret keys of the trapdoor permutation, respectively, K is the secret key of function F, **T**, **N** are maps and m is the maximum number of the values in our range queries. The map **N** is used to store the pair keyword/(ST_c, c) (current search token and the counter c, please see Algorithm 2 for more details.) and is kept by the client. The map **T** is the encrypted database EDB that used to store the encrypted indices which is kept by the server.

Search($[a, b], \sigma, m,$ EDB): The protocol is executed between a client and a server. The client asks for documents, whose keywords are in the range $[a, b]$, where $0 \leq a \leq b < m$. The current state of EDB is σ and the integer m describes the maximum number of values. Note that knowing m, the client can easily construct the complete binary tree. The server returns a collection of file indices of requested documents.

Update($add, v, ind, \sigma, m,$ EDB): The protocol is performed jointly by a client and server. The client wishes to add an integer v together with a file index ind to EDB. The state of EDB is σ, the number of values m. There are following three cases:

- $v < m$: The client simply adds ind to the leaf, which contains value v and its parents (See line 9-24 in Algorithm 2). This is a basic update, which is similar to the one from [20].
- $v = m$: The client first updates the complete binary tree to which she adds the value v. If a new root is added to the new complete binary tree, then the server needs to add all file indices of the old complete binary tree to the new one. Finally, the server needs to add ind to the leaf, which contains value v and its parents.
- $v > m$: The client uses **Update** as many times as needed. For simplicity, we only present the simple case $v = m$, i.e., the newly added value v equals the maximum number of values of the current range $[0, m-1]$, in the description of Algorithm 2.

The DSSE supports range queries at the cost of large client storage since the number of search tokens is linear in the number of all nodes of the current tree instead of only leaves. In [20],

Algorithm 2 Construction A

Setup(1^λ)
Input security parameter 1^λ
Output $(\texttt{TPK}, \texttt{TSK}, K, \mathbf{T}, \mathbf{N}, m)$
1: $K \leftarrow \{0,1\}^\lambda$
2: $(\texttt{TSK}, \texttt{TPK}) \leftarrow \texttt{TKeyGen}(1^\lambda)$
3: **T, N** $\leftarrow$ empty map
4: $m = 0$
5: **return** $(\texttt{TPK}, \texttt{TSK}, K, \mathbf{T}, \mathbf{N}, m)$

Search($[a,b], \sigma, m,$ EDB)
Client:
Input $[a,b], \sigma, m$
Output (K_n, ST_c, c)
1: $\texttt{CBT} \leftarrow$ **TCon**(m)
2: $\texttt{ABT} \leftarrow$ **TAssign**(CBT)
3: **RSet** $\leftarrow$ Find the minimum nodes to cover $[a,b]$ in ABT
4: **for** $n \in$ **RSet do**
5: $K_n \leftarrow F_K(n)$
6: $(ST_c, c) \leftarrow \mathbf{N}[n]$
7: **if** $(ST_c, c) \neq \perp$ **then**
8: Send (K_n, ST_c, c) to the server.
9: **end if**
10: **end for**
Server:
Input (K_n, ST_c, c), EDB
Output (ind)
11: Upon receiving (K_n, ST_c, c)
12: **for** $i = c$ to 0 **do**
13: $UT_i \leftarrow H_1(K_n, ST_i)$
14: $e \leftarrow \mathbf{T}[UT_i]$
15: $ind \leftarrow e \oplus H_2(K_n, ST_i)$
16: Output the ind
17: $ST_{i-1} \leftarrow \Pi(\texttt{TPK}, ST_i)$
18: **end for**

Update($add, v, ind, \sigma, m,$ EDB)
Client:
Input add, v, ind, σ, m
Output (UT_{c+1}, e)
1: $\texttt{CBT} \leftarrow$ **TCon**(m)
2: **if** $v = m$ **then**
3: CBT$\leftarrow$**TUpdate**($add, v,$ CBT)
4: $m \leftarrow m + 1$
5: **if** CBT added a new root **then**
6: $(ST_c, c) \leftarrow \mathbf{N}[\texttt{root}_o]$
7: $\mathbf{N}[\texttt{root}_n] \leftarrow (ST_c, c)$
8: **end if**
9: Get the leaf n_v of value v.
10: $\texttt{ABT} \leftarrow$ **TAssign**(CBT)
11: **NSet** $\leftarrow$ **TGetNodes**($n_v,$ ABT)
12: **for** every node $n \in$ **NSet do**
13: $K_n \leftarrow F_K(n)$
14: $(ST_c, c) \leftarrow \mathbf{N}[n]$
15: **if** $(ST_c, c) = \perp$ **then**
16: $ST_0 \leftarrow \mathcal{M}, c \leftarrow -1$
17: **else**
18: $ST_{c+1} \leftarrow \Pi^{-1}(\texttt{TSK}, ST_c)$
19: **end if**
20: $\mathbf{N}[n] \leftarrow (ST_{c+1}, c+1)$
21: $UT_{c+1} \leftarrow H_1(K_n, ST_{c+1})$
22: $e \leftarrow ind \oplus H_2(K_n, ST_{c+1})$
23: Send (UT_{c+1}, e) to the Server.
24: **end for**
25: **else if** $v < m$ **then**
26: Execute line 9-24.
27: **end if**
Server:
Input (UT_{c+1}, e), EDB
Output EDB
28: Upon receiving (UT_{c+1}, e)
29: Set $\mathbf{T}[UT_{c+1}] \leftarrow e$

the number of entries at the client is $|W|$, while it would be roughly $2|W|$ in this construction. Moreover, the communication cost is high since the server needs to return all file indices to the client for every search. To overcome the weakness, we give a new construction with lower client storage and communication cost in the following section.

3.4.4 DSSE Range Queries - Construction B

In this section, we give the second construction by leveraging the Paillier cryptosystem [22] and bit string representation, which significantly reduce the client storage and communication cost compared with the first one at the cost of losing forward privacy. With the homomorphic addition property of the Paillier cryptosystem, we can add and delete the file indices by parsing them into binary strings, as illustrated in Section 3.4.2. Next, we briefly describe our second DSSE, which can not only support range queries but also achieve backward privacy. The scheme is described in Algorithm 3.

Setup(1^λ): For a security parameter 1^λ , the algorithm returns ($\mathtt{PK}, \mathtt{SK}, K,$ **T**$, m$), where $\mathtt{PK}$ and $\mathtt{SK}$ are the public and secret keys of the Paillier cryptosystem, respectively, K is the secret key of a PRF F, m is the maximum number of values which can be used to reconstruct the binary tree and the encrypted database $\mathtt{EDB}$ is stored in a map **T** which is kept by the server.

Search($[a, b], \sigma,\ m,\ \mathtt{EDB}$): The protocol is executed between a client and a server. The client queries for documents, whose keywords are in the range $[a, b]$, where $0 \leq a \leq b < m$. σ is the state of $\mathtt{EDB}$, and integer m specifies the maximum values for our range queries. The server returns encrypted file indices e to the client, who can decrypt e by using the secret key $\mathtt{SK}$ of Pailler Cryptosystem and obtain the file indices of requested documents.

Update($op, v, ind, \sigma,\ m,\ \mathtt{EDB}$): The protocol runs between a client and a server. A requested update is named by the parameter op. The integer v and the file index ind specifies the tree nodes that need to be updated. The current state σ, the integer m and the server with input $\mathtt{EDB}$. If $op = add$, the client generates a bit string as prescribed in Section 3.4.2. In case when $op = delete$, the client creates the complement bit string as given in Section 3.4.2. The bit string bs is encrypted using the Paillier cryptosystem. The encrypted string is denoted by e. There are following three cases:

- $v < m$: The client sends the encrypted bit string e with the leaf n_v containing value v and its parents to server. Next the server adds e with the existing encrypted bit strings corresponding to the nodes specified by the client. See line 11-23 in Algorithm 3 which is similar to the update in Algorithm 2.

- $v = m$: The client first updates the complete binary tree to which she adds the value v. If a new root is added to the new complete binary tree, then the client retrieves the encrypted bit string of the root (before the update). Next, the client adds it to the new root by sending it with the new root to the server. Finally, the client adds e to the leaf that contains value v and its parents as in $v < m$ case.

- $v > m$: The client uses **Update** as many times as needed. For simplicity, we only consider $v = m$, where m is the number of values in the maximum range.

Algorithm 3 Construction B

Setup(1^λ)
Input security parameter 1^λ
Output $(\mathtt{PK}, \mathtt{SK}, K, \mathbf{T}, m)$
$K \leftarrow \{0,1\}^\lambda$
$(\mathtt{SK}, \mathtt{PK}) \leftarrow \mathtt{KeyGen}(1^\lambda)$
$\mathbf{T} \leftarrow$ empty map
$m = 0$
return $(\mathtt{PK}, \mathtt{SK}, K, \mathbf{T}, m)$

Search($[a, b], \sigma, m, \mathtt{EDB}$)
Client:
Input $[a, b], \sigma, m$
Output (UT_n)
$\mathtt{CBT} \leftarrow$ **TCon**(m)
$\mathtt{ABT} \leftarrow$ **TAssign**$(\mathtt{CBT})$
RSet $\leftarrow$ Find the minimum nodes to cover $[a, b]$ in $\mathtt{ABT}$
for $n \in$ **RSet do**
    $UT_n \leftarrow F_K(n)$
    Send UT_n to the server.
end for
Server:
Input (UT_n), $\mathtt{EDB}$
Output (e)
Upon receiving UT_n
$e \leftarrow \mathbf{T}[UT_n]$
Send e to the Client.

Update($op, v, ind, \sigma, m, \mathtt{EDB}$)
Client:
Input op, v, ind, σ, m
Output (UT_n, e)
$\mathtt{CBT} \leftarrow$ **TCon**(m)
if $v = m$ **then**
    $\mathtt{CBT} \leftarrow$ **TUpdate**$(add, v, \mathtt{CBT})$
    $m \leftarrow m + 1$
    if CBT added a new root **then**
        $UT_{\mathtt{root}_o} \leftarrow F_K(\mathtt{root}_o)$
        $UT_{\mathtt{root}_n} \leftarrow F_K(\mathtt{root}_n)$
        $e \leftarrow \mathbf{T}[UT_{\mathtt{root}_o}]$
        $\mathbf{T}[UT_{\mathtt{root}_n}] \leftarrow e$
    end if
    Get the leaf n_v of value v.
    $\mathtt{ABT} \leftarrow$ **TAssign**$(\mathtt{CBT})$
    NSet $\leftarrow$ **TGetNodes**$(n_v, \mathtt{ABT})$
    if $op = add$ **then**
        Generate the bit string bs as state in Bit String Representation of Section 3.4.2.
    else if $op = del$ **then**
        Generate the complement bit string bs as state in Bit String Representation of Section 3.4.2.
    end if
    for every node $n \in$ **NSet do**
        $UT_n \leftarrow F_K(n)$
        $e \leftarrow \mathtt{Enc}(\mathtt{PK}, bs)$
        Send (UT_n, e) to the server.
    end for
else if $v < m$ **then**
    Execute line 11-23.
end if
Server:
Input (UT_n, e), $\mathtt{EDB}$
Output $\mathtt{EDB}$
Upon receiving (UT_n, e)
$e' \leftarrow \mathbf{T}[UT_n]$
if $e' \neq \perp$ **then**
    $e \leftarrow e \cdot e'$
end if
$\mathbf{T}[UT_n] \leftarrow e$

In this construction, it achieves backward privacy. Moreover, the communication overhead between the client and the server is significantly reduced due to the fact that for each query, the server returns a single ciphertext to the client at the cost of supporting a small number of documents. Since, in Paillier cryptosystem, the length of the message is usually small and fixed (e.g., 1024 bits).

This construction can be applied to applications where the number of documents is small, and simultaneously the number of keywords can be large. The reason for this is the fact that for a given keyword, the number of documents that contain it is small. Con-

sider a temperature forecast system that uses a database, which stores record from different sensors (IoT) located in different cities across Australia. In the application, the cities (sensors) can be considered as documents, and temperature measurements can be considered as the keywords. For example, Sydney and Melbourne have a temperature of 18°C. Adelaide and Wollongong have got 17°C and 15°C, respectively. If we query for cities whose temperature measurements are in the range from 17 to 18°C, then the outcome includes Adelaide, Sydney, and Melbourne. Here, the number of cities (documents) is not large. The number of different temperature measurements (keywords) can be large, depending on requested precision.

3.5 Security Analysis

Similar to [13], for a range query $q = [a, b]$, let $\{n_{c_1}, ..., n_{c_t}\}$ be the tree cover of interval $[a, b]$. We consider n_{c_i} as a keyword and parse a range query into several keywords. Before define the leakage functions, we define a search query $q = (t, [a, b]) = \{(t, n_{c_1}), \cdots, (t, n_{c_t})\}$. For an update query, if we want to update a file ind with value v, we may need to update the corresponding leaf node and its parents in the tree denoted as $\{n_{u_1}, \cdots, n_{u_t}\}$. We define an update query $u = (t, op, (v, ind)) = \{(t, op, (n_{u_1}, ind)), \cdots, (t, op, (n_{u_t}, ind))\}$. For a list of search query $Q = \{(t, n) : (t, n) \in \{q\}\}$ and a list of update query $Q' = \{(t, op, (n, ind)) : (t, op, (n, ind)) \in \{u\}\}$ Then, following [20], the leakage to the server is summarized as follows:

- Search pattern $\mathtt{sp}(n) = \{t : (t, n) \in Q\}$, it leaks the timestamp t that the same search query on n.
- History $\mathtt{Hist}(n) = \{(t, op, ind) : (t, op, (n, ind)) \in Q'\}$, the history of keyword n. It includes all the updates made to $\mathtt{DB}(n)$ and when the update happened.
- contain pattern $\mathtt{cp}(n) = \{t' : \mathtt{DB}(n) \subseteq \mathtt{DB}(n') \text{ and } t' < t, (t', n'), (t, n) \in Q\}$, it leaks the time t' of previous search query on keyword n', where $\mathtt{DB}(n) \subseteq \mathtt{DB}(n')$. Note that, $\mathtt{cp}(n)$ is an inherited leakage for range queries when the file indices are revealed to the server. If a query n is a subrange of query n', then the file index set for n will also be a subset of the file index set for n'.

3.5.1 Forward Privacy

Following [20], forward privacy means that an update does not leak any information about keywords of updated documents matching a query we previously issued. A formal definition is given below:

Definition 2. *([20]) A $\mathcal{L}$-adaptively-secure DSSE scheme Γ is forward-private if the update leakage*

function $\mathcal{L}^{Updt}$ can be written as

$$\mathcal{L}^{Updt}(op, in) = \mathcal{L}'(op, (ind_i, \mu_i))$$

where (ind_i, μ_i) is the set of modified documents paired with number μ_i of modified keywords for the updated document ind_i.

3.5.2 Construction A

Since the first DSSE construction is based on [20], it inherits the security of the original design. Adaptive security of construction A can be proven in the Random Oracle Model and is a modification of the security proof of [20].

Theorem 1. *(Adaptive forward privacy of A). Let $\mathcal{L}_{\Gamma_A} = (\mathcal{L}^{Srch}_{\Gamma_A}, \mathcal{L}^{Updt}_{\Gamma_A})$, where $\mathcal{L}^{Srch}_{\Gamma_A}(n)$ $= (sp(n), \mathtt{Hist}(n), cp(n))$, $\mathcal{L}^{Updt}_{\Gamma_A}(add, n, ind) = \perp$. The construction A is $\mathcal{L}_{\Gamma_A}$-adaptively forward private.*

Compared with [20], this construction additionally leaks the contain pattern $\mathtt{cp}$ as described in Section 3.3.1. Other leakages are exactly the same as [20]. Since the server executes one keyword search and updates one keyword/file-index pair at a time. Note that the server does not know the secret key of the trapdoor permutation, so it cannot learn anything about the pair even if the keyword has been searched by the client previously.

Proof. As mentioned before, we parse a range interval into several keywords. Following [20], we will set a serial of games from $\mathtt{DSSEREAL}^{\Gamma_A}_{\mathcal{A}}(1^\lambda)$ to $\mathtt{DSSEIDEAL}^{\Gamma_A}_{\mathcal{A},\mathcal{S}_1}(1^\lambda)$.

Game $G_{1,0}$: $G_{1,0}$ is exactly same as the real world game $\mathtt{DSSEREAL}^{\Gamma_A}_{\mathcal{A}}(1^\lambda)$.

$$\Pr[\mathtt{DSSEREAL}^{\Gamma_A}_{\mathcal{A}}(1^\lambda) = 1] = \Pr[G_{1,0} = 1].$$

Game $G_{1,1}$: Instead of calling F when generating k_n, $G_{1,1}$ picks a new random key when it inputs a new keyword n, and stores it in a table $\mathtt{Key}$ so it can be reused next time. If an adversary $\mathcal{A}$ is able to distinguish between $G_{1,0}$ and $G_{1,1}$, we can then build a reduction to distinguish between F and a truly random function. More formally, there exists an efficient adversary $\mathcal{B}_1$ such that

$$\Pr[G_{1,0} = 1] - \Pr[G_{1,1} = 1] \leq \mathtt{Adv}^{\mathtt{prf}}_{F,\mathcal{B}_1}(\lambda).$$

Game $G_{1,2}$: In $G_{1,2}$, we pick random strings in replace of calling hash function H_1, where H_1 is modeled as a random oracle. For every search, the output of H_1 is programmed where $H_1(K_n, ST_c(n)) = \mathtt{UT}[n, c]$.

Algorithm 4 Game $G_{1,2}$ and single box for $G'_{1,2}$

Setup(1^λ)

$(\mathtt{TSK}, \mathtt{TPK}) \leftarrow \mathtt{TKeyGen}(1^\lambda)$
T, **N** $\leftarrow$ empty map
$m = 0$
$bad \leftarrow false$
return $(\mathtt{TPK}, \mathtt{TSK}, K, \mathbf{T}, \mathbf{N}, m)$

Search$([a, b], \sigma; m,$ EDB)

Client:

CBT $\leftarrow$ **TCon**(m)
ABT $\leftarrow$ **TAssign**(CBT)
RSet $\leftarrow$ **TGetCover**$([a, b],$ ABT)
for $n \in$ **RSet do**
  $K_n \leftarrow \mathtt{Key}(n)$
  $(ST_0, \cdots, ST_c, c) \leftarrow \mathbf{N}[n]$
  if $(ST_c, c) \neq \perp$ **then**
    for $i = 0$ to c **do**
      $H_1(K_n, ST_i) \leftarrow \mathtt{UT}[n, i]$
    end for
    Send (K_n, ST_c, c) to the server.
  end if
end for

Server:

Upon receiving (K_n, ST_c, c)
for $i = c$ to 0 **do**
  $UT_i \leftarrow H_1(K_n, ST_i)$
  $e \leftarrow \mathbf{T}[UT_i]$
  $ind \leftarrow e \oplus H_2(K_n, ST_i)$
  Output the ind
  $ST_{i-1} \leftarrow \Pi(\mathtt{TPK}, ST_i)$
end for

Update$(add, v, ind, \sigma; m,$ EDB)

Client:

CBT $\leftarrow$ **TCon**(m)
if $v = m$ **then**
  CBT$\leftarrow$**TUpdate**$(add, v,$ CBT)
  $m \leftarrow m + 1$
  if CBT added a new root **then**
    $(ST_c, c) \leftarrow \mathbf{N}[root_{old}]$
    $\mathbf{N}[root_{new}] \leftarrow (ST_c, c)$
  end if
  Get the leaf n_v of value v.
  ABT $\leftarrow$ **TAssign**(CBT)
  NSet $\leftarrow$ **TGetNodes**$(n_v,$ ABT)
  for every node $n \in$ **NSet do**
    $K_n \leftarrow \mathtt{Key}(n)$
    $(ST_c, c) \leftarrow \mathbf{N}[n]$
    if $(ST_c, c) = \perp$ **then**
      $ST_0 \leftarrow \mathcal{M}, c \leftarrow -1$
    else
      $ST_{c+1} \leftarrow \Pi^{-1}(\mathtt{TSK}, ST_c)$
    end if
    $\mathbf{N}[n] \leftarrow (ST_0, \cdots, ST_{c+1}, c+1)$
    $UT_{c+1} \leftarrow \{0,1\}^x$
    if $H_1(K_n, ST_{c+1} \neq \perp)$ **then**
      $\boxed{bad \leftarrow true, UT_{c+1} \leftarrow H_1(K_n, ST_{c+1})}$
    end if
    $\mathtt{UT}[n, c+1] \leftarrow UT_{c+1}$
    $e \leftarrow ind \oplus H_2(K_n, ST_{c+1})$
    Send (UT_{c+1}, e) to the Server.
  end for
else if $v < m$ **then**
  Execute line 9-24.
else
  We can use **Update** many times.
end if

Server:

Upon receiving (UT_{c+1}, e)
Set $\mathbf{T}[UT_{c+1}] \leftarrow e$

$H_1(k, v)$

$v' \leftarrow H_1(k, v)$
if $v' = \perp$ **then**
  $v' \leftarrow \{0,1\}^x$
  if $\exists n, c$ s.t $v = ST_c \in \mathbf{N}[n]$ **then**
    $\boxed{bad \leftarrow true, v' \leftarrow UT[n, c]}$
  end if
  $H_1(k, v) \leftarrow v'$
end if
return v'

Algorithm 4 describes this game and introduce an intermediate game $G'_{1,2}$. $G'_{1,2}$ is used to keep the consistency of H_1's transcript. In **Update**, we chooses a random value for $UT_c(n)$ and stores it in table UT, and programed it to the output of (K_n, ST_c) in **Search**.

Since H_1's outputs in $G'_{1,2}$ and $G_{1,1}$ are perfectly indistinguishable, so we have

$$\Pr[G_{1,1} = 1] = \Pr[G'_{1,2} = 1].$$

$G'_{1,2}$ and $G_{1,2}$ are also perfectly identical unless the bad happens (set to $true$). $\Pr[G'_{1,2} = 1] - \Pr[G_{1,2} = 1] \leq \Pr[bad \text{ is set to true in } G'_{1,2}]$

Following [20], the possibility for $H_1(K_n, ST_{c+1})$ already exists is the advantage of breaking the one-wayness of the trapdoor permutation which is $\mathtt{Adv}^{\mathtt{OW}}_{\Pi,\mathcal{B}_2}(1^\lambda)$. Assume the query make N queries, then we have

$$\Pr[G_{1,1} = 1] - \Pr[G_{1,2} = 1] = \Pr[G'_{1,2} = 1] - \Pr[G_{1,2} = 1] \leq N \cdot \mathtt{Adv}^{\mathtt{OW}}_{\Pi,\mathcal{B}_2}(1^\lambda).$$

Game $G_{1,3}$: Similar to $G_{1,2}$, $G_{1,3}$ programs H_2. The same steps can be reused, giving that there is an adversary B_3, such that

$$\Pr[G_{1,2} = 1] - \Pr[G_{1,3} = 1] \leq N \cdot \mathtt{Adv}^{\mathtt{OW}}_{\Pi,\mathcal{B}_3}(1^\lambda).$$

Note that, we can consider $B_2 = B_3$ without loss of generality.

Game $G_{1,4}$: In $G_{1,4}$, we keep the records of the random generated encrypted strings of the H_1 and H_2. In **Update**, we choose random values for update tokens and ciphertexts in Table $\mathtt{UT}$ and $\mathtt{e}$, respectively. Then we program them identically to the outputs of the corresponding hash functions in **Search**. Then $G_{1,4}$ is exactly same as the $G_{1,3}$. More formally,

$$\Pr[G_{1,4} = 1] = \Pr[G_{1,3} = 1]$$

Simulator $\mathcal{S}_1$ With the contain pattern $\mathtt{cp}$, the simulator can reuse the certain update token UT to simulate the inclusion relationship between the keywords. We can use the search pattern $\hat{n} \leftarrow \min \mathtt{sp}(n)$ and history $\mathtt{Hist}$ to simulate the **Search** and **Update**. Similar to [20], we have

$$\Pr[G_{1,4} = 1] = \Pr[\mathtt{DSSEIDEAL}^{\Gamma_A}_{\mathcal{A},\mathcal{S}_1}(1^\lambda) = 1]$$

Finally,

$$\Pr[\mathtt{DSSEREAL}^{\Gamma_A}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathtt{DSSEIDEAL}^{\Gamma_A}_{\mathcal{A},\mathcal{S}_1}(1^\lambda) = 1]$$

$$\leq \mathtt{Adv}^{\mathtt{prf}}_{F,\mathcal{B}_1}(1^\lambda) + 2N \cdot \mathtt{Adv}^{\mathtt{OW}}_{\Pi,\mathcal{B}_2}(1^\lambda)$$

which completes the proof. □

3.5.3 Backward Privacy

Backward privacy means that a search query on keyword n does not leak the file indices that previously added and later deleted. More formally, we modify the Type I definition of

Algorithm 5 Simulator $\mathcal{S}_1$

$\mathcal{S}$.**Setup**(1^λ)

$(\mathtt{TSK}, \mathtt{TPK}) \leftarrow \mathtt{TKeyGen}(1^\lambda)$
N, **T** $\leftarrow$ empty map
$t = 0$
return $(\mathtt{TPK}, \mathtt{TSK}, \mathbf{T}, \mathbf{N})$

$\mathcal{S}$.**Update**()

Client:

$\mathtt{UT}[t] \leftarrow \{0,1\}^x$
$\mathtt{e}[t] \leftarrow \{0,1\}^y$
Send $(\mathtt{UT}[t], \mathtt{e}[t])$ to the server.
$t \leftarrow t + 1$

$\mathcal{S}$.**Search**$(\mathtt{sp}(n), \mathtt{Hist}(n), \mathtt{cp}(n))$

Client:

$\hat{n} \leftarrow \min \mathtt{sp}(n)$
$K_{\hat{n}} \leftarrow \mathtt{Key}[\hat{n}]$
Parse $\mathtt{cp}(n)$ as t'
if $t' \neq \perp$ **then**
    Get the c-th search token ST_c of previously queried keyword at time t'.
else
    Parse $\mathtt{Hist}(n)$ as $((t_0, add, ind_0), \cdots, (t_c, add, ind_c))$
    if $\mathtt{Hist}(n) = \perp$ **then**
        return $\varnothing$
    end if
    for $i = 0$ to c **do**
        Set $H_1(K_{\hat{n}}, ST_i) \leftarrow \mathtt{UT}[t_i]$
        Set $H_2(K_{\hat{n}}, ST_i) \leftarrow \mathtt{e}[t_i] \oplus ind_i$
        $ST_{i+1} \leftarrow \Pi^{-1}_{\mathtt{TSK}}(ST_i)$
    end for
end if
Send $(K_{\hat{n}}, ST_c)$ to the server.

[21]. It leaks keyword n has been updated[7], the total number of updates on n. More formally,

Definition 3. *A $\mathcal{L}$-adaptively-secure DSSE scheme Γ is backward-private if the the search and update leakage functions $\mathcal{L}^{Srch}$, $\mathcal{L}^{Updt}$ can be written as $\mathcal{L}^{Updt}(n) = \mathcal{L}'(n)$, $\mathcal{L}^{Srch} = \mathcal{L}''(sp(n))$.*

3.5.4 Construction B

The adaptive security of the second DSSE construction relies on the semantic security of the Paillier cryptosystem. All file indices are encrypted using the public key of the Paillier cryptosystem. Without the secret key, the server cannot learn anything from the ciphertext. Note that, during the update, this construction leaks which keyword has been updated.

Theorem 2. *(Adaptive backward privacy of B). Let $\mathcal{L}_{\Gamma_B} = (\mathcal{L}^{Srch}_{\Gamma_B}, \mathcal{L}^{Updt}_{\Gamma_B})$, where $\mathcal{L}^{Srch}_{\Gamma_B}(n) = (sp(n))$, $\mathcal{L}^{Updt}_{\Gamma_B}(op, n, ind) = (n)$. Construction B is $\mathcal{L}_{\Gamma_B}$-adaptively backward-private.*

During the update, the construction B leaks which keyword has been updated. However, it does not leak the type of update (either add or del) on encrypted file indices because both addition and deletion are achieved by homomorphic addition. Moreover, it does not leak contain pattern $\mathtt{cp}$ and the file indices that previously added and later deleted since the file indices have been encrypted, and the server can learn nothing without the secret key.

Proof. For Theorem 2, we also set a serial of games from $\mathtt{DSSEREAL}^{\Gamma_B}_{\mathcal{A}}(1^\lambda)$ to $\mathtt{DSSEIDEAL}^{\Gamma_B}_{\mathcal{A},\mathcal{S}_2}(1^\lambda)$.

[7] Instead of leaking the keyword n in the plaintext form; it may be leaked in the masked form.

Game $G_{2,0}$: $G_{2,0}$ is exactly same as the real world game $\texttt{DSSEREAL}_{\mathcal{A}}^{\Gamma_B}(1^\lambda)$.

$$\Pr[\texttt{DSSEREAL}_{\mathcal{A}}^{\Gamma_B}(1^\lambda) = 1] = \Pr[G_{2,0} = 1]$$

Game $G_{2,1}$: Instead of calling F when generating UT_n, $G_{2,1}$ picks a new random key when it inputs a new keyword n, and stores it in a table $\texttt{Key}$ so it can be reused next time. If an adversary $\mathcal{A}$ is able to distinguish between $G_{2,0}$ and $G_{2,1}$, we can then build a reduction able to distinguish between F and a truly random function. More formally, there exists an efficient adversary $\mathcal{B}_1$ such that

$$\Pr[G_{2,0} = 1] - \Pr[G_{2,1} = 1] \leq \texttt{Adv}_{F,\mathcal{B}_1}^{\texttt{prf}}(1^\lambda).$$

Game $G_{2,2}$: We replace the bit string bs with a all 0 bit string. If an adversary $\mathcal{A}$ is able to distinguish between $G_{2,1}$ and $G_{2,2}$, we can then build an adversary $\mathcal{B}_2$ to break the semantic security of Paillier cryptosystem. More formally, there exists an efficient adversary $\mathcal{B}_2$ such that

$$\Pr[G_{2,1} = 1] - \Pr[G_{2,2} = 1] \leq \texttt{Adv}_{\Sigma,\mathcal{B}_2}^{\texttt{IND-CPA}}(1^\lambda).$$

Algorithm 6 Simulator $\mathcal{S}_2$

$\mathcal{S}$.**Setup**(1^λ)

1: $(\texttt{SK}, \texttt{PK}) \leftarrow \texttt{KeyGen}(1^\lambda)$
2: **return** $(\texttt{PK}, \texttt{SK}, t)$

$\mathcal{S}$.**Update**(n)

Client:

1: $UT_n \leftarrow \texttt{Key}(n)$
2: $e \leftarrow \texttt{Enc}(\texttt{PK}, 0 \cdots 0)$
3: Send (UT_n, e) to the server.

$\mathcal{S}$.**Search**$(\texttt{sp}(n))$

Client:

1: $\hat{n} \leftarrow \min \texttt{sp}(n)$
2: $UT_{\hat{n}} \leftarrow Key(\hat{n})$
3: Send $UT_{\hat{n}}$ to the server.

Simulator Now, we can simulator the $\texttt{DSSEIDEAL}$ with the leakage functions defined in this Theorem. We removed the useless part which will not influence the client's transcript. See Algorithm 6 for more details. This two games is indistinguishable. So we have

$$\Pr[G_{2,2} = 1] = \Pr[\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}_2}^{\Gamma_B}(1^\lambda) = 1]$$

Finally,

$$\Pr[\texttt{DSSEREAL}_{\mathcal{A}}^{\Gamma_B}(1^\lambda) = 1] - \Pr[\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}_2}^{\Gamma_B}(1^\lambda) = 1]$$
$$\leq \texttt{Adv}_{F,\mathcal{B}_1}^{\texttt{prf}}(1^\lambda) + \texttt{Adv}_{\Sigma,\mathcal{B}_2}^{\texttt{IND-CPA}}(1^\lambda)$$

which completes the proof.

□

3.6 Conclusion

In this chapter, we give two secure DSSE schemes that support range queries. The first DSSE construction applies our binary tree to Bost [20]'s framework, which achieves forward privacy. However, it incurs a large storage overhead in the client and a high communication cost between the client and the server. To achieve backward privacy, we propose the second DSSE construction with range queries by applying Paillier cryptosystem and bit string representation. In this construction, we use the fixed update token to reduce the client and the server storage at the cost of losing forward privacy. In addition, it can not support a large number of documents. Although the second DSSE construction cannot support a large number of documents, it can still be very useful in certain applications.

Chapter 4

Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy

Dynamic Searchable Symmetric Encryption (DSSE) enables a client to perform updates and searches on encrypted data, which makes it very useful in practice. To protect DSSE from the leakage of updates (leading to break query or data privacy), two new security notions, forward and backward privacy, have been proposed recently. Although extensive attention has been paid to forward privacy, this is not the case for backward privacy. Backward privacy, first formally introduced by Bost et al., is classified into three types from weak to strong, exactly Type-III to Type-I. To the best of our knowledge, however, no practical DSSE schemes without trusted hardware (e.g., SGX) have been proposed so far, in terms of the strong backward privacy and constant roundtrips between the client and the server.

In this chapter, we present a new DSSE scheme by leveraging simple symmetric encryption with homomorphic addition and bitmap index. The new scheme can achieve both forward and backward privacy with one roundtrip. In particular, the backward privacy we achieve in our scheme (denoted by Type-I^{-}) is somewhat stronger than Type-I. Moreover, our scheme is very practical as it involves only lightweight cryptographic operations. To make it scalable for supporting billions of files, we further extend it to a multi-block setting. Finally, we give the corresponding security proofs and experimental evaluation, which demonstrate both the security and practicality of our schemes, respectively.

4.1 Introduction

Cloud storage solutions become increasingly popular and economically attractive for users who need to handle large volumes of data. To protect the data stored on the cloud, users normally encrypt the data before sending it to the cloud. Unfortunately, encryption destroys the natural structure of data, and consequently, data needs to be decrypted before processing. To solve this dilemma, searchable symmetric encryption (SSE) has been proposed [10–12]. SSE not only protects the confidentiality of data but also permits searching over encrypted data without a need for decryption. Furthermore, SSE is much more efficient compared to other cryptographic techniques such as oblivious RAM (ORAM) that attract a punishing computational overhead [52, 53].

Early SSE solutions were designed for a static setting, i.e., an encrypted database cannot be updated. This feature of SSE severely restricts their applications. To overcome this limitation and make SSE practical, dynamic searchable symmetric encryption (DSSE) was

proposed (see [15, 17]). DSSE allows both searching and updating. However, security analysis becomes more complicated as an adversary can observe the behavior of the database during the updates (addition and deletion of data). For instance, an adversary can find out if an added/deleted file contains previously searched keywords. Cash et al. [16] argued that updates could leak information about the contents of the database as well as search queries and keywords involved. For example, file-injection attacks can reveal user queries by adding to a database a small number of carefully designed files [18].

Consequently, two new security notions called forward and backward privacy were proposed to deal with the leakages mentioned above. They were informally introduced by Stefanov et al. in 2014 [19]. Roughly speaking, for any adversary who may continuously observe the interactions between the server and the client, forward privacy is satisfied if the addition of new files does not leak any information about previously queried keywords. In a similar vein, backward privacy holds if files that previously added and later deleted do not leak "too much" information within any period that two search queries on the same keyword happened[1]. Bost [20] formally defined forward privacy and designed a forward-private DSSE scheme, which is resistant against file-injection attacks [18]. The scheme has been extended by Zuo et al. [54], so it supports range queries. In contrast, backward privacy attracted less attention. Recently, Bost et al. [21] defined three variants of backward privacy in order from strong to weak. They are:

- Type-I – backward privacy with insertion pattern. Given a keyword w and a time interval between two search queries on w, then Type-I leaks information about when new files containing w were inserted and the total number of updates on w.
- Type-II – backward privacy with update pattern. Apart from the leakages of Type-I, it additionally leaks when all updates (including deletion) related to w occurred.
- Type-III – weak backward privacy. It leaks information of Type-II, and it also leaks exactly when a previous addition has been canceled by which deletion.

For example, assume that a query has the following form {time, operation, (keyword, file)}. Given the following queries: $\{1, search, w\}$, $\{2, add, (w, f_1)\}$, $\{3, add, (w, f_2)\}$, $\{4, add, (w, f_3)\}$, $\{5, del, (w, f_2)\}$ and $\{6, search, w\}$. Then after time 6, Type-I leaks that there are 4 updates, the files f_1 and f_3 match the keyword w, and these two files were added at time 2 and 4, respectively. Type-II additionally leaks time 3 and 5 when the updates related to keyword w occurred. Type-III also leaks the fact that the addition at time 3 has been canceled by the deletion at time 5[2].

[1]The files are leaked if the second search query is issued after the files are added but before they are deleted. This is unavoidable since the adversary can easily tell the difference of the search results before and after the same search query.

[2]In this example, there is only one addition/deletion pair. For Type-II, the server knows which addition has been canceled by which deletion easily. However, there may have many addition/deletion pairs; then the server cannot know which deletion cancels which addition.

Bost et al. [21] gave several constructions with different security/efficiency trade-offs. Their FIDES scheme achieves Type-II backward privacy. Their schemes DIANA_{del} and Janus provide better performance at the expense of security (they are Type-III backward-private). Their scheme MONETA, which is based on the recent TWORAM construction of Garg et al. [53], achieves Type-I backward privacy. Ghareh Chamani et al. [55], however, argued that the MONETA scheme is highly impractical due to the fact that it is based on TWORAM, and it serves mostly as a theoretical result for the feasibility of Type-I schemes. Sun et al. [56] proposed a new DSSE scheme named Janus++. It is more efficient than Janus as it is based on symmetric puncturable encryption. Janus++ can only achieve the same security level as Janus (Type-III).

Very recently, Ghareh Chamani et al. [55] designed three DSSE schemes. The first scheme MITRA achieves Type-II backward privacy, and it is based on symmetric key encryption getting better performance than FIDES [21]. The second scheme ORION achieves Type-I backward privacy. It requires $O(logN)$ rounds of interaction and applies ORAM [52], where N is the total number of keyword/file-identifier pairs. The third design is HORUS. The number of interactions is reduced to $O(d_w)$ at the expense of lower security guarantees (Type-III backward privacy), where d_w is the number of deleted entries for w. Zuo et al. [54] also constructed two DSSE schemes supporting range queries. Their first scheme achieves forward privacy. Their second scheme (called SchemeB) uses bit string representation and the Paillier cryptosystem, which achieves backward privacy. However, they did not provide any formal analysis for the backward privacy of their scheme. To the best of our knowledge, no practical DSSE schemes achieve both the high-level backward privacy and constant interactions between the client and the server.

Our Contributions. In this chapter, we propose an efficient DSSE scheme (named FB-DSSE) with a stronger backward privacy (denoted as Type-I$^-$ backward privacy) and one roundtrip (without considering the retrieval of actual files), which also achieves forward privacy. This scheme is based on a bitmap index and a simple symmetric encryption with homomorphic addition. Later, we extend it to a multi-block setting (named MB-FB-DSSE). Table 4.1 compares our schemes (FB-DSSE and MB-FB-DSSE) with other designs supporting backward privacy. In particular, our contributions are as follows:

- We formally introduce a new type of backward privacy, named Type-I$^-$ backward privacy. It does not leak the insertion time of each matched files which is somewhat stronger than Type-I. More precisely, for a query with a keyword w, it only leaks the number of previous updates associated with w, time when these updates happened, and files that currently match w. Type-I$^-$ leaks no information about when each file was inserted. For our example, Type-I$^-$ only leaks that time 2, 3, 4, 5 are updates and f_1, f_3 currently matching keyword w[3]. Although it is not clear the impact of leaking the

[3]Note that, it does not leak the insertion time of f_1 and f_3.

Table 4.1: Comparison with previous works

Scheme	Roundtrips bet. Client and Server	Client Storage	Forward Privacy	Backward Privacy	Without ORAM
FIDES [21]	2	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-II	✓
DIANA_{del} [21]	2	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✓
Janus [21]	1	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✓
Janus++ [56]	1	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✓
MITRA [55]	2	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-II	✓
HORUS [55]	$O(logd_w)$	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-III	✗
SchemeB [54]	2	$O(\|\mathbf{W}\|log\|D\|)$	✗	Unknown	✓
MONETA [21]	3	$O(1)$	✓	Type-I	✗
ORION [55]	$O(logN)$	$O(1)$	✓	Type-I	✗
Our schemes	1	$O(\|\mathbf{W}\|log\|D\|)$	✓	Type-I$^-$	✓

N is the number of keyword/file-identifier pairs, d_w is the number of deleted entries for keyword w. $|\mathbf{W}|$ is the collection of distinct keywords, $|D|$ is the total number of files.

insertion time in practice, it is believed that the less information the scheme leaks, the higher security it guarantees, since the leakage might be leveraged by the adversary to launch some potential attacks.

- We design a Type-I$^-$ backward-private DSSE FB−DSSE by leveraging the bitmap index and the simple symmetric encryption with homomorphic addition. FB−DSSE also achieves forward privacy, which is based on the framework of [20]. In the scheme, we achieve forward privacy through a new technique that deploys symmetric primitive instead of the public primitive [20] (one-way trapdoor permutation), which makes our scheme more efficient.

- To support an even larger number of files with improved efficiency, we extend our first scheme to the multi-block setting. We call it MB−FB−DSSE. In our experimental analysis, for the MB−FB−DSSE scheme with 1 billion files, the search and update time are 5.84s and 46.41ms, respectively, where the number of blocks is 10^3, and the bit length of each block is 10^6. For the same number of files, the FB−DSSE scheme consumes 9.07s for search and 125.23ms for update (note that, bit length is 10^9). Finally, the security analyses are given to show that our schemes are forward and Type-I$^-$ backward private.

Remark: We note that our scheme does not leak the insertion time of each matched file for a search query w, but leaks the update time for each update. Therefore, it achieves somewhat stronger security than Type-I, but strictly stronger security than Type-II. In the conference version [54], we claim that it achieves stronger security than Type-I, named Type-I$^-$. In this full version, we correct the inaccurate definition for Type-I$^-$ in Section 3.4 of the conference version [54] and the corresponding example given in "**Our Contributions**". In general, to eliminate the update time of each update for a search query w, it is always required to use

some sort of ORAM technique [52, 55] to touch every item in a database at the expense of low efficiency. In this chapter, we focus on achieving stronger backward privacy without using ORAM.

4.1.1 Related Work

For the completeness and consistency of this Chapter, we enumerate some related works that already appeared in Section 2.1.1 and 3.1.1. Song et al. [10] showed how to perform a keyword search over encrypted data using symmetric encryption. To search for a keyword w, the server compares every encrypted keyword in a file with a token (issued by the client). The search time is linear with the number of keyword/file-identifier pairs, which is not efficient. Later, Curtmola et al. [11] designed an efficient SSE based on the inverted index, which achieves sub-linear search time. The authors also quantified the leakage of an SSE and gave a formal security definition for SSE. Cash et al. [12] proposed a highly scalable SSE, which supports large databases. Following this work, many SSE schemes have been proposed addressing different aspects. For example, Sun et al. [37] focused on the usage of SSE in a multi-client setting. Zuo et al. [14] proposed an SSE scheme, which supports more general Boolean queries. To support database updates, dynamic SSE schemes are introduced in [15, 17, 20, 21].

Early schemes have been designed under the assumption that the encrypted database is static; i.e., it cannot be updated. Dynamic SSE schemes were introduced in [15, 17]. For DSSE schemes, it is assumed that an encrypted database can be updated, i.e., new files can be added, and some existing files can be removed. However, the dynamic nature of databases brings new security problems. Two security notions, namely, forward and backward privacy, have been informally introduced in [19]. Further works concentrate on refinements of the privacy notions for DSSE schemes [20, 21, 55, 56].

There is also a line of investigation that concentrates on the design of SSE schemes that can handle richer (complex) queries. Cash et al. [12] proposed an SSE scheme that handles Boolean queries. Faber et al. [13] extended the scheme so it can handle more complex queries about ranges, substrings, and wild cards. A majority of forward private DSSE schemes support single keyword queries only. Zuo et al. [54] proposed a forward private DSSE scheme supporting range queries. Recently, SGX has been used to instantiate hardware-based SSE. We refer readers to [57, 58] for more details.

4.1.2 Organization

The remaining sections of this chapter are organized as follows. In Section 4.2, we give the necessary background information and describe building blocks that are used in this chapter. In Section 4.3, we define DSSE and its security notions. In Section 4.4, we present our DSSE schemes. Their security and experimental analyses are given in Section 4.5 and

Section 4.6, respectively. Finally, Section 4.7 concludes this chapter.

4.2 Preliminaries

In this chapter, λ denotes the security parameter, $||$ stands for the concatenation and $|m|$ denotes the bit length of m. We use bitmap index[4] to represent file identifiers [59]. More precisely, there is a bit string bs of the length ℓ, where ℓ is the maximum number of files that a scheme can support. The i-th bit of bs is set to 1 if there exists file f_i, and 0 otherwise. Fig. 4.1 illustrates an instance for 6 files, i.e. $\ell = 6$. Assume that there exists file f_2 and f_3 (see Fig. 4.1.(a)). If we want to add file f_1, we need to generate the bit string $2^1 = 000010$ and add it to the original bit string (see Fig. 4.1.(b)). Now, if we want to delete file f_2, we need to generate the bit string $-2^2 = -4 = -000100$. As our index computation is done modulo 2^6, we can convert $-4 = 2^6 - 2^2 = 60 \pmod{2^6}$, which is 111100 in binary. The string 111100 is added to the original bit string (see Fig. 4.1.(c)). Note that manipulation on bitmap indexes for addition and deletion can be done by modulo addition. In other words, the bitmap index can be (homomorphically) encrypted and updated (to reflect addition or deletion of files) using encryption with the homomorphic property.

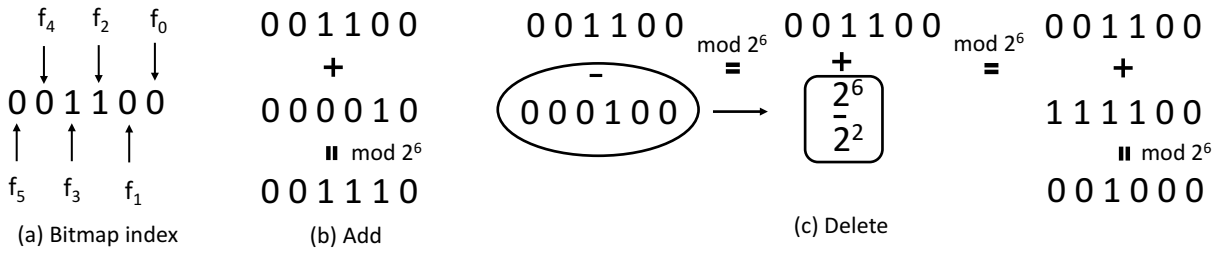


Figure 4.1: An example of our bitmap index

4.2.1 Simple Symmetric Encryption with Homomorphic Addition

A simple symmetric encryption with homomorphic addition Π [60] consists of the following four algorithms `Setup`, `Enc`, `Dec` and `Add` as described below:

- $n \leftarrow \texttt{Setup}(1^\lambda)$: For the security parameter λ, it outputs a public parameter n, where $n = 2^\ell$ is the message space and ℓ is the maximum number of files a scheme can support.

[4] A special kind of data structure which has been widely used in the database community.

- $c \leftarrow \texttt{Enc}(sk, m, n)$: For a message m, the public parameter n and a random secret key sk ($0 \leq sk < n$), it computes a ciphertext $c = sk + m \bmod n$, where m is the message $0 \leq m < n$. Note that, for every encryption, the secret key sk needs to be stored, and it can only be used once.

- $m \leftarrow \texttt{Dec}(sk, c, n)$: For the ciphertext c, the public parameter n and the secret key sk, it recovers the message $m = c - sk \bmod n$.

- $\hat{c} \leftarrow \texttt{Add}(c_0, c_1, n)$: For two ciphertexts c_0, c_1 and the public parameter n, it computes $\hat{c} = c_0 + c_1 \bmod n$, where $c_0 \leftarrow \texttt{Enc}(sk_0, m_0, n)$, $c_1 \leftarrow \texttt{Enc}(sk_1, m_1, n)$, $n \leftarrow \texttt{Setup}(1^\lambda)$ and $0 \leq sk_0, sk_1 < n$.

Correctness. For the correctness of this scheme, it is required that sum of two ciphertexts $\hat{c} = c_0 + c_1 \bmod n$ decrypts to $m_0 + m_1 \bmod n$ under the $\hat{sk} = sk_0 + sk_1 \bmod n$ or in other words

$$\texttt{Dec}(\hat{sk}, \hat{c}, n) = \hat{c} - \hat{sk} \bmod n = m_0 + m_1 \bmod n.$$

It is easy to check that this requirement holds.

Remark. For the encryption and decryption algorithms of Π, the secret key sk can only be used once.

Perfect Security [60]. We say Π is perfectly secure if for any PPT adversary $\mathcal{A}$, their advantage in the perfect-security game is negligible or

$$\mathbf{Adv}^{\mathsf{PS}}_{\Pi,\mathcal{A}}(\lambda) = |\Pr[\mathcal{A}(\texttt{Enc}(sk, m_0, n)) = 1] - \Pr[\mathcal{A}(\texttt{Enc}(sk, m_1, n)) = 1]| \leq \epsilon,$$

where $n \leftarrow \texttt{Setup}(1^\lambda)$, the secret key sk ($0 \leq sk < n$) is kept secret and $\mathcal{A}$ chooses m_0, m_1 s.t. $0 \leq m_0, m_1 < n$.

4.2.2 Notations

Notations used in the chapter are given in Table 4.2.

4.3 DSSE Definition and Security Model

A database $\texttt{DB}$ is a list of file-identifier/keyword-set pairs, which is denoted as $\texttt{DB} = (f_i, \mathbf{W}_i)_{i=1}^{\ell}$. The file identifier is $f_i \in \{0,1\}^\lambda$, $\mathbf{W}_i \subseteq \{0,1\}^*$ is a set of all keywords contained in the file f_i and ℓ is the total number of files in $\texttt{DB}$. We also denote $\mathbf{W} = \cup_{i=1}^{\ell} \mathbf{W}_i$ as all keywords in $\texttt{DB}$. We identify $\mathbf{W}$ as a collection of all distinct keywords that occur in $\texttt{DB}$. Note that, $|\mathbf{W}|$ is the total number of keywords and $N = \sum_{i=1}^{\ell} |\mathbf{W}_i|$ is denoted as the total

Table 4.2: Notations (used in our schemes)

DB	A database
λ	The security parameter
ST_c	The current search token for a keyword w
EDB	The encrypted database EDB which is a map
F	A secure PRF
W	The set of all keywords of the database DB
CT	A map stores the current search token ST_c and counter c for every keyword in **W**
f_i	The i-th file
bs	The bit string which is used to represent the existence of files
ℓ	The length of bs
e	The encrypted bit string
Sum_e	The sum of the encrypted bit strings
sk	The one time secret key
Sum_{sk}	The sum of the one time secret keys
B	The number of blocks
bs	The bit string array with length B
e	The encrypted bit string array with length B
$\mathbf{Sum_e}$	The sum of the encrypted bit string arrays with length B
sk	The one time secret key array with length B
$\mathbf{Sum_{sk}}$	The sum of the one time secret key arrays with length B

number of file-identifier/keyword pairs. A set of files that satisfy a query q is denoted by DB(q). Note that, in this chapter, we use bitmap index to represent the file identifiers. For a search query q, the result is a bit string bs, which represents a list of file identifiers in DB(q). For an update query u, a bit string bs is used to update a list of file identifiers. Moreover, we isolate the actual files from the metadata (e.g. file identifiers). We only focus on the search of the metadata. The ways we can retrieve the encrypted files are not described in this chapter.

4.3.1 DSSE Definition

A DSSE scheme consists of an algorithm **Setup** and two protocols **Search** and **Update** that are executed between a client and a server. They are described as follows:

- (EDB, σ) $\leftarrow$ **Setup**(1^λ, DB): For a security parameter λ and a database DB, the algorithm outputs a pair: an encrypted database EDB and a state σ. EDB is stored by the server and σ is kept by the client.
- ($\mathcal{I}$, $\perp$) $\leftarrow$ **Search**(q, σ; EDB): For a state σ, the client issues a query q and interacts with the server who holds EDB. At the end of the protocol, the client outputs a set of file identifiers $\mathcal{I}$ that match q and the server outputs nothing.
- (σ', EDB$'$) $\leftarrow$ **Update**(σ, op, in; EDB): For a state σ, the operation $op \in \{add, del\}$ and a collection of $in = (f, \mathbf{w})$ pairs, the client requests the server (who holds EDB) to update

database by adding/deleting files specified by the collection in. Finally, the protocol returns an updated state σ' to the client and an updated encrypted database $\texttt{EDB}'$ to the server.

Remark. There are two variants of the result model for SSE schemes. In the first one (considered in the work [12]), the server returns encrypted file identifiers $\mathcal{I}$, so the client needs to decrypt them. In the second one (studied in the work [20]), the server returns the file identifiers to the client directly. In our work, we consider the first variant, where the protocol returns encrypted file identifiers.

4.3.2 Security Model

DSSE security is modeled by interaction between the Real and Ideal worlds called $\texttt{DSSEREAL}$ and $\texttt{DSSEIDEAL}$, respectively. The behavior of $\texttt{DSSEREAL}$ is exactly the same as the original DSSE. However, $\texttt{DSSEIDEAL}$ reflects a behavior of a simulator $\mathcal{S}$, which takes the leakage of the original DSSE as input. The leakage is defined by a function $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$, which details what information the adversary $\mathcal{A}$ can learn during execution of the **Setup** algorithm, **Search** and **Update** protocols.

If the adversary $\mathcal{A}$ can distinguish $\texttt{DSSEREAL}$ from $\texttt{DSSEIDEAL}$ with a negligible advantage, the information leakage is limited to $\mathcal{L}$ only. More formally, we consider the following security game. The adversary $\mathcal{A}$ interacts with one of the two worlds $\texttt{DSSEREAL}$ or $\texttt{DSSEIDEAL}$ and would like to guess it.

- $\texttt{DSSEREAL}_{\mathcal{A}}(\lambda)$: First **Setup**$(\lambda, \texttt{DB})$ is run and the adversary gets $\texttt{EDB}$. $\mathcal{A}$ performs search queries q (or update queries (op, in)). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.
- $\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda)$: Simulator $\mathcal{S}$ with the input $\mathcal{L}^{Setup}(\lambda, \texttt{DB}))$ is executed. For search queries q (or update queries (op, in)) generated by the adversary $\mathcal{A}$, the simulator $\mathcal{S}$ replies by using the leakage function $\mathcal{L}^{Search}(q)$ (or $\mathcal{L}^{Update}(op, in)$). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.

Definition 4. *Given a DSSE scheme and the security game described above. The scheme is $\mathcal{L}$-adaptively-secure if for every PPT adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ (with the input $\mathcal{L}$) such that,*

$$|\Pr[\texttt{DSSEREAL}_{\mathcal{A}}(\lambda) = 1] - \Pr[\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda).$$

Leakage Function. Before define the leakage function, we define a search query $q = (t, w)$ and an update query $u = (t, op, (w, bs))$, where t is the timestamp, w is the keyword to be searched (or updated), op is the update operation and bs denotes a list of file identifiers to be

updated. For a list of search queries Q, we define a search pattern $\mathtt{sp}(w) = \{t : (t, w) \in Q\}$, where t is a timestamp. The search pattern leaks the repetition of search queries on w. Result pattern $\mathtt{rp}(w) = \overline{bs}$, $\overline{bs}$ represents all file identifiers that currently matching w. Note that, after a search query, we implicitly assume that the server knows the final result $\overline{bs}$[5].

4.3.3 Forward Privacy

Informally, for any adversary who may continuously observe the interactions between the server and the client, forward privacy guarantees that an update does not leak information about the newly added files that match the previously issued queries. The definition given below is taken from [20]:

Definition 5. *A $\mathcal{L}$-adaptively-secure DSSE scheme is forward-private if the update leakage function $\mathcal{L}^{Update}$ can be written as*

$$\mathcal{L}^{Update}(op, in) = \mathcal{L}'(op, \{(f_i, \mu_i)\}),$$

where $\{(f_i, \mu_i)\}$ is the set of modified file-identifier/keywords pairs, μ_i is the number of keywords corresponding to the updated file f_i.

Remark. In this chapter, the leakage function will be $\mathcal{L}^{Update}(op, w, bs) = \mathcal{L}'(op, bs)$.

4.3.4 Backward Privacy

Similarly, within any period that two search queries on the same keyword happened, backward privacy ensures that it does not leak information about the files that have been previously added and later deleted. Note that information about files is leaked if the second search query is issued after the files are added but before they are deleted. In 2017, Bost et al. [21] formulated three different levels of backward privacy from Type-I to Type-III in decreasing the level of privacy. In our construction, we use a new data structure (see Fig. 4.1), which achieves a stronger level of backward privacy. We call it Type-I$^-$, which is somewhat stronger than Type-I. We refer readers to [21] for more details. Type-I$^-$ and Type-I definitions are given below.

- Type-I$^-$: Given a time interval between two search queries for a keyword w, then it leaks the files that currently match w and the total number of updates for w and the update time for each update.
- Type-I: Given a time interval between two search queries for a keyword w, then it leaks not only files that currently match w and the total number of updates for w but additionally when the matched files were inserted.

[5]The client may retrieve the file identifiers represented by $\overline{bs}$ which is not described in this chapter.

To define Type-I$^-$ formally, we need a new leakage functions $\texttt{Time}$. For a search query on keyword w, $\texttt{Time}(w)$ lists the timestamp t of all updates corresponding to w. Formally, for a sequence of update queries Q':

$$\texttt{Time}(w) = \{t : (t, op, (w, bs)) \in Q'\}.$$

Definition 6. *A $\mathcal{L}$-adaptively-secure DSSE scheme is Type-I$^-$ backward-private iff the search and update leakage function $\mathcal{L}^{Search}$, $\mathcal{L}^{Update}$ can be written as:*

$$\mathcal{L}^{Update}(op, w, bs) = \mathcal{L}'(op), \mathcal{L}^{Search}(w) = \mathcal{L}''(\texttt{sp}(w), \texttt{rp}(w), \texttt{Time}(w)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

4.4 Our Construction

In this section, we give our Type-I$^-$ backward private DSSE scheme. To achieve forward privacy, we follow the framework of the forward-private DSSE from [20]. To improve the efficiency of the underlying forward-private DSSE, we use a hash function that replaces a public key primitive (i.e., one-way function in [20]) to achieve forward privacy. See Section 4.4.2 for more details.

4.4.1 Overview

To achieve backward privacy, the DSSE schemes from [21, 56] used puncturable encryption, which can be used to "puncture" the deleted file identifiers. Then the deleted file identifiers cannot be decrypted (searched). The schemes achieve Type-III backward privacy only. In our construction, instead of encrypting file identifiers independently, we use a new data structure, the bitmap index (as illustrated in Fig. 4.1.(a)), where all file identifiers are represented by a single bit string. To add or delete a file identifier, the bit string is modified as shown in Fig. 4.1.(b) and Fig. 4.1.(c), respectively. Besides, our scheme does not leak the update type since both addition and deletion are done by one modulo addition[6]. To securely update the encrypted database, our scheme requires additive homomorphic encryption as the underlying encryption primitive.

The most popular additive homomorphic encryption is the Paillier cryptosystem [22]. Unfortunately, the Paillier cryptosystem attracts a very large computation overhead and can support a limited number of files (up to the number of bits in a message/ciphertext space, e.g., 1024 bits). After observing our bitmap index, we notice that we only need the addition of ciphertexts (the bit strings). Also, we do not need to use one key for all encryption and decryption. Therefore we can use simple symmetric encryption (the key can be only used

[6]Deletion is by adding a negative number.

once) with homomorphic addition (Section 4.2.1) to add the ciphertexts (and the keys) simultaneously. To save the client storage, we can use a hash function with a secret key K to generate all the one time keys. E.g., $H(K, c)$, where c is a counter. It is worth noting that this technique has been used in [60] in the context of wireless sensor networks.

4.4.2 DSSE with Forward and Stronger Backward Privacy

Now we are ready to give our forward and stronger backward private DSSE construction $\texttt{FB-DSSE}$ – see Algorithm 7. Our scheme is based on the framework of the forward private DSSE from [20], a simple symmetric encryption with homomorphic addition $\Pi = (\texttt{Setup}, \texttt{Enc}, \texttt{Dec}, \texttt{Add})$, and a keyed PRF F_K with key K. The scheme is defined by the following algorithms:

- $(\texttt{EDB}, \sigma = (n, K, \textbf{CT})) \leftarrow \textbf{Setup}(1^\lambda)$: The algorithm is run by a client. It takes the security parameter λ as input. Then it chooses a secret key K and an integer n, where $n = 2^\ell$ and ℓ is the maximum number of files that this scheme can support. Moreover, it initializes two empty maps $\texttt{EDB}$ and **CT**, which are used to store the encrypted database as well as the current search token ST_c and the current counter c (the number of updates) for each keyword $w \in \textbf{W}$, respectively. Finally, it outputs encrypted database $\texttt{EDB}$ and the state $\sigma = (n, K, \textbf{CT})$, and the client keeps $(K, \textbf{CT})$ secret.

- $(\sigma', \texttt{EDB}') \leftarrow \textbf{Update}(w, bs, \sigma; \texttt{EDB})$: The algorithm runs between a client and a server. The client inputs a keyword w, a state σ, and a bit string bs[7]. Next, the client encrypts the bit string bs by using the simple symmetric encryption with homomorphic addition to get the encrypted bit string e. To save the client storage, the one time key sk_c is generated by a hash function $H_3(K'_w, c)$, where c is the counter. Then he/she chooses a random search token and uses a hash function to get the update token. He/She also uses another hash function to mask the previous search token. After that, the client sends the update token, e, and the masked previous search token C to the server and update **CT** to get a new state σ'. Finally, the server outputs an updated encrypted database $\texttt{EDB}'$.

- $bs \leftarrow \textbf{Search}(w, \sigma; \texttt{EDB})$: The protocol runs between a client and a server. The client inputs a keyword w and a state σ, and the server inputs $\texttt{EDB}$. Firstly, the client gets the search token corresponding to the keyword w from **CT** and generates the K_w. Then he/she sends them to the server. The server retrieves all the encrypted bit strings e corresponding to w. To reduce the communication overhead, the server adds them together by using the homomorphic addition ($\texttt{Add}$) of the simple symmetric encryption to get the final result Sum_e and sends it to the client. Finally, the client decrypts it and outputs the final bit string bs, which can be used to retrieve the matching files. Note

[7] Note that, we can update many file identifiers through one update query by using bit string representation bs.

Algorithm 7 FB-DSSE

Setup(1^λ)

$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
CT, EDB $\leftarrow$ empty map
return $(\texttt{EDB}, \sigma = (n, K, \textbf{CT}))$

Update$(w, bs, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \textbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
    $c \leftarrow -1$, $ST_c \leftarrow \{0,1\}^\lambda$
end if
$ST_{c+1} \leftarrow \{0,1\}^\lambda$
$\textbf{CT}[w] \leftarrow (ST_{c+1}, c+1)$
$UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
$C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
$sk_{c+1} \leftarrow H_3(K'_w, c+1)$
$e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to server.

Server:

$\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(w, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \textbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
    return $\varnothing$
end if
Send (K_w, ST_c, c) to server.

Server:

$Sum_e \leftarrow 0$
for $i = c$ to 0 **do**
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $(e_i, C_{ST_{i-1}}) \leftarrow \texttt{EDB}[UT_i]$
    $Sum_e \leftarrow \texttt{Add}(Sum_e, e_i, n)$
    Remove $\texttt{EDB}[UT_i]$
    if $C_{ST_{i-1}} = \perp$ **then**
        $Break$
    end if
    $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
end for
$\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
Send Sum_e to client.

Client:

$Sum_{sk} \leftarrow 0$
for $i = c$ to 0 **do**
    $sk_i \leftarrow H_3(K'_w, i)$
    $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum_e, n)$
return bs

that, in order to save the server storage, for every search, the server can remove all entries corresponding to w and store the final result Sum_e corresponding to the current search token ST_c to the EDB.

4.4.3 Multi-block Extension for Large Number of Files

The number of files supported by FB-DSSE is determined by the length of the public parameter $n = 2^\ell$, which is the modulus. Theoretically, it can be of an arbitrary length, but a larger n (e.g., $\ell = 2^{23}$) will significantly slow down modular operations. Efficiency analysis and experiments will be given in Section 4.6. However, there are many applications that require a system able to manage up to a billion files. Therefore, we still need to find an efficient solution for such applications.

In this section, we extend our basic scheme to multi-block setting in order to handle a larger number of files efficiently. The idea is to split the long bit sequence ℓ into multiple smaller blocks and have multiple (e.g. B) blocks **bs** instead of one block bs. For every block of **bs**, the operations are exactly the same as FB-DSSE. We denote the extension of FB-DSSE as MB-FB-DSSE, which is shown in Algorithm 8. To ease the understanding of this

algorithm, we put the differences of MB−FB−DSSE from FB−DSSE into boxes. This scheme consists of following algorithms:

- $(\mathtt{EDB}, \sigma = (n, K, \mathbf{CT})) \leftarrow \mathbf{Setup}(1^\lambda)$: The algorithm is exactly same as the one in FB−DSSE.

Algorithm 8 Multi-block extension MB−FB−DSSE (Differences in boxes)

Setup(1^λ)

1: Same as the one in FB−DSSE.

Update$(w, \mathbf{bs}, \sigma; \mathtt{EDB})$

Client:

1: Same as the one in FB−DSSE.
2: **for** $j = 0$ to B **do**
3: $\boxed{\mathbf{sk}_{c+1}[j] \leftarrow H_3(K'_w, c+1||j)}$
4: $\boxed{\mathbf{e}_{c+1}[j] \leftarrow \mathtt{Enc}(\mathbf{sk}_{c+1}[j], \mathbf{bs}[j], n)}$
5: **end for**
6: Send $(ST_{c+1}, (\boxed{\mathbf{e}_{c+1}}, C_{ST_c}))$ to the server.

Server:

7: $\mathtt{EDB}[ST_{c+1}] \leftarrow (\boxed{\mathbf{e}_{c+1}}, C_{ST_c})$

Search$(w, \sigma; \mathtt{EDB})$

Client:

1: Same as the one in FB−DSSE.

Server:

2: $\boxed{\mathbf{Sum_e} \leftarrow 0}$
3: **for** $i = c$ to $0\boxed{, j = 0 \text{ to } B}$ **do**
4: $UT_i \leftarrow H_1(K'_w, ST_i)$
5: $\boxed{(\mathbf{e}_i, C_{ST_{i-1}}) \leftarrow \mathtt{EDB}[UT_i]}$
6: $\boxed{\mathbf{Sum_e}[j] \leftarrow \mathtt{Add}(\mathbf{Sum_e}[j], \mathbf{e}_i[j], n)}$
7: Same as the one in FB−DSSE.
8: **end for**
9: $\boxed{\mathtt{EDB}[ST_c] \leftarrow (\mathbf{Sum_e}, \perp)}$
10: Send $\boxed{\mathbf{Sum_e}}$ to the client.

Client:

11: $\boxed{\mathbf{Sum_{sk}} \leftarrow 0}$
12: **for** $i = c$ to $0\boxed{, j = 0 \text{ to } B}$ **do**
13: $\boxed{\mathbf{sk}_i[j] \leftarrow H_3(K'_w, i||j)}$
14: $\boxed{\mathbf{Sum_{sk}}[j] \leftarrow \mathbf{Sum_{sk}}[j] + \mathbf{sk}_i[j] \bmod n}$
15: **end for**
16: **for** $\boxed{j = 0 \text{ to } B}$ **do**
17: $\boxed{\mathbf{bs}[j] \leftarrow \mathtt{Dec}(\mathbf{Sum_{sk}}[j], \mathbf{Sum_e}[j], n)}$
18: **end for**
19: **return bs**

- $(\sigma', \mathtt{EDB}') \leftarrow \mathbf{Update}(w, \mathbf{bs}, \sigma; \mathtt{EDB})$ and $\mathbf{bs} \leftarrow \mathbf{Search}(w, \sigma; \mathtt{EDB})$: The two protocols are similar to the ones in FB−DSSE. The difference is that we use multiple blocks **bs** rather than one block bs to support large number of files, where **bs** is an array with length B and it stores all the bit strings bs. For each block, the operations are exactly same as the ones in FB−DSSE. Correspondingly, we have **e**, **sk**, $\mathbf{Sum_e}$ and $\mathbf{Sum_{sk}}$ which are arrays with the length of B. $\mathbf{Sum_e}$ and $\mathbf{Sum_{sk}}$ are used to store the sum of all encrypted bit string arrays **e** and secret keys **sk**, respectively. The length of the bit string bs will be reduced and the computation time of these blocks will be shorter.

4.5 Security Analysis

In this section, we give the security analysis of our schemes.

Theorem 3. *(Adaptive security of FB−DSSE). Let F be a secure PRF, $\Pi = (\mathtt{Setup}, \mathtt{Enc}, \mathtt{Dec}, \mathtt{Add})$ be a perfectly secure simple symmetric encryption with homomorphic addition, and H_1, H_2 and H_3 be random oracles. We define $\mathcal{L}_{\mathtt{FB-DSSE}} = (\mathcal{L}^{Search}_{\mathtt{FB-DSSE}}, \mathcal{L}^{Update}_{\mathtt{FB-DSSE}})$, where $\mathcal{L}^{Search}_{\mathtt{FB-DSSE}}(w) = (\mathtt{sp}(w), \mathtt{rp}(w), \mathtt{Time}(w))$ and $\mathcal{L}^{Update}_{\mathtt{FB-DSSE}}(op, w, bs) = \perp$. Then FB−DSSE is $\mathcal{L}_{\mathtt{FB-DSSE}}$-adaptively secure.*

Similar to [20], we will set a set of games from DSSEREAL to DSSEIDEAL, and we will show that every two consecutive games is indistinguishable. Finally, we will simulate DSSEIDEAL with the leakage functions defined in **Theorem 3**.

Proof. In this proof, the server is the adversary $\mathcal{A}$ who tries to break the security of our FB−DSSE. The challenger $\mathcal{C}$ is responsible for generating the search tokens and ciphertexts, and the simulator $\mathcal{S}$ simulates the transcripts between $\mathcal{A}$ and $\mathcal{C}$ at the end.

Game G_0: G_0 is exactly same as the real world game $\mathtt{DSSEREAL}_{\mathcal{A}}^{\mathtt{FB-DSSE}}(\lambda)$, such that

$$\Pr[\mathtt{DSSEREAL}_{\mathcal{A}}^{\mathtt{FB-DSSE}}(\lambda) = 1] = \Pr[G_0 = 1].$$

Game G_1: In G_1, when querying F to generate a key for a keyword w, the challenger $\mathcal{C}$ chooses a new random key if the keyword w is never queried before, and stores it in a table Key. Otherwise, return the key corresponding to w in the table Key. If an adversary $\mathcal{A}$ is able to distinguish between G_0 and G_1, we can then build an adversary $\mathcal{B}_1$ to distinguish between F and a truly random function. More formally,

$$\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq \mathbf{Adv}_{F,\mathcal{B}_1}^{\mathtt{prf}}(\lambda).$$

Game G_2: In G_2, as depicted in Algorithm 9, in the **Update** protocol, we pick random strings for the update token UT and store it in table UT. Then, in the **Search** protocol, we program these random strings to the output of the random oracle H_1 where $H_1(K_w, ST_c) = \mathtt{UT}[w, c]$. When $\mathcal{A}$ queries H_1 with the input (K_w, ST_c), $\mathcal{C}$ will output $\mathtt{UT}[w, c]$ to $\mathcal{A}$ and store this entry in table $\mathtt{H}_1$ for future queries. If the entry (K_w, ST_{c+1}) already in table $\mathtt{H}_1$, $\mathtt{UT}[w, c+1]$ cannot be programed to the output of $H_1(K_w, ST_{c+1})$ and this game aborts. Now, we will show that the possibility of the game aborts is negligible. The search token is chosen randomly by the challenger $\mathcal{C}$, then the possibility that the adversary guesses the right search token ST_{c+1} is $1/2^\lambda$. Assume $\mathcal{A}$ makes polynomial p queries, then the possibility is $p/2^\lambda$. So we have

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq p/2^\lambda$$

Game G_3: In G_3, we model the H_2 as a random oracle which is similar to H_1 in G_2. Then we have

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq p/2^\lambda$$

Game G_4: In G_4, similar to G_2, we model the H_3 as a random oracle. $\mathcal{A}$ does not know the key K'_w, then the possibility that he guesses the right key is $1/2^\lambda$ (we set the length of K'_w to λ). Assume $\mathcal{A}$ makes polynomial p queries, the possibility is $p/2^\lambda$. So we have

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \leq p/2^\lambda$$

Algorithm 9 G_2 for $\texttt{FB-DSSE}$

Setup(1^λ)

$K \xleftarrow{\$} \{0,1\}^\lambda, n \leftarrow \texttt{Setup}(1^\lambda)$
CT, $\texttt{EDB} \leftarrow$ empty map
return $(\texttt{EDB}, \sigma = (n, K, \mathbf{CT}))$

Update$(w, bs, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow \texttt{Key}(w)$
$(ST_0, \cdots, ST_c, c) \leftarrow \mathbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
  $c \leftarrow -1$, $ST_c \leftarrow \{0,1\}^\lambda$
end if
$ST_{c+1} \leftarrow \{0,1\}^\lambda$
$\mathbf{CT}[w] \leftarrow (ST_0, \cdots, ST_{c+1}, c+1)$
$UT_{c+1} \leftarrow \{0,1\}^\lambda$
$\texttt{UT}[w, c+1] \leftarrow UT_{c+1}$
$C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
$sk_{c+1} \leftarrow H_3(K'_w, c+1)$
$e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to server.

Server:

$\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(w, \sigma; \texttt{EDB})$

Client:

$K_w || K'_w \leftarrow \texttt{Key}(w)$
$(ST_0, \cdots, ST_c, c) \leftarrow \mathbf{CT}[w]$
if $(ST_c, c) = \perp$ **then**
  return $\varnothing$
end if
for $i = 0$ to c **do**
  $\texttt{H}_1(K_w, ST_i) \leftarrow \texttt{UT}[w, i]$
end for
Send (K_w, ST_c, c) to server.

Server:

$Sum_e \leftarrow 0$
for $i = c$ to 0 **do**
  $UT_i \leftarrow H_1(K_w, ST_i)$
  $(e_i, C_{ST_{i-1}}) \leftarrow \texttt{EDB}[UT_i]$
  $Sum_e \leftarrow \texttt{Add}(Sum_e, e_i, n)$
  Remove $\texttt{EDB}[UT_i]$
  if $C_{ST_{i-1}} = \perp$ **then**
    $Break$
  end if
  $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
end for
$\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
Send Sum_e to client.

Client:

$Sum_{sk} \leftarrow 0$
for $i = c$ to 0 **do**
  $sk_i \leftarrow H_3(K'_w, i)$
  $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum_e, n)$
return bs

Game G_5: In G_5, we replace the bit string bs with an all 0 bit string, and the length of the all 0 bit string is ℓ. If an adversary $\mathcal{A}$ is able to distinguish between G_5 and G_4, then we can build a reduction $\mathcal{B}_2$ to break the perfect security of the simple symmetric encryption with homomorphic addition Π. So we have

$$\Pr[G_4 = 1] - \Pr[G_5 = 1] \leq \mathbf{Adv}^{\texttt{PS}}_{\Pi, \mathcal{B}_2}(\lambda).$$

Simulator Now we can replace the searched keyword w with $\texttt{sp}(w)$ in G_5 to simulate the simulator $\mathcal{S}$ in Algorithm 10, $\mathcal{S}$ uses the first timestamp $\hat{w} \leftarrow \texttt{min sp}(w)$ for the keyword w. We remove the useless part of Algorithm 9 which will not influence the view of $\mathcal{A}$.

Now we are ready to show that G_5 and **Simulator** are indistinguishable. For **Update**, it is obvious since we choose new random strings for each update in G_5. For **Search**, $\mathcal{S}$ starts from the current search token ST_c and choose a random string for previous search token. Then $\mathcal{S}$ embeds it to the ciphertext C through H_2. Moreover, $\mathcal{S}$ embeds the $\overline{bs}$ to the ST_c and

all 0s to the remaining search tokens through H_3. Finally, we map the pairs (w, i) to the globe update count t. Then we can map the values in table $\mathtt{UT}$, $\mathtt{C}$ and $\mathtt{sk}$ that we chose randomly in **Update** to the corresponding values for the pair (w, i) in the **Search**. Hence,

$$\Pr[G_5 = 1] = \Pr[\mathtt{DSSEIDEAL}^{\mathtt{FB\text{-}DSSE}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]$$

Finally,

$$\Pr[\mathtt{DSSEREAL}^{\mathtt{FB\text{-}DSSE}}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathtt{DSSEIDEAL}^{\mathtt{FB\text{-}DSSE}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]$$

$$\leq \mathbf{Adv}^{\mathtt{prf}}_{F,\mathcal{B}_1}(\lambda) + \mathbf{Adv}^{\mathtt{PS}}_{\Pi,\mathcal{B}_2}(\lambda) + 3p/2^{\lambda}$$

which completes the proof.

Algorithm 10 Simulator $\mathcal{S}$ for $\mathtt{FB\text{-}DSSE}$

$\mathcal{S}$.**Setup**(1^{λ})

1: $n \leftarrow \mathtt{Setup}(1^{\lambda})$
2: **CT**, $\mathtt{EDB} \leftarrow$ empty map
3: **return** $(\mathtt{EDB}, \mathbf{CT}, n)$

$\mathcal{S}$.**Update**()

Client:

1: $\mathtt{UT}[t] \leftarrow \{0,1\}^{\lambda}$
2: $\mathtt{C}[t] \leftarrow \{0,1\}^{\lambda}$
3: $\mathtt{sk}[t] \leftarrow \{0,1\}^{\lambda}$
4: $\mathtt{e}[t] \leftarrow \mathtt{Enc}(\mathtt{sk}[t], 0s, n)$
5: Send $\mathtt{UT}[t], (\mathtt{e}[t], \mathtt{C}[t]))$ to the server.
6: $t \leftarrow t + 1$

$\mathcal{S}$.**Search**$(\mathtt{sp}(w), \mathtt{rp}(w), \mathtt{Time}(w))$

Client:

1: $\hat{w} \leftarrow \mathtt{min}\ \mathtt{sp}(w)$
2: $K_{\hat{w}} || K'_{\hat{w}} \leftarrow \mathtt{Key}(\hat{w})$
3: $(ST_c, c) \leftarrow \mathbf{CT}[\hat{w}]$
4: parse $\mathtt{rp}(w)$ as $\overline{bs}$.
5: Parse $\mathtt{Time}(w)$ as $(t_0, \cdots, t_c)$.
6: **if** $(ST_c, c) = \perp$ **then**
7: **return** $\varnothing$
8: **end if**
9: **for** $i = c$ to 0 **do**
10: $ST_{i-1} \leftarrow \{0,1\}^{\lambda}$
11: Program $H_1(K_{\hat{w}}, ST_i) \leftarrow \mathtt{UT}[t_i]$
12: Program $H_2(K_{\hat{w}}, ST_i) \leftarrow \mathtt{C}[t_i] \oplus ST_{i-1}$
13: **if** $i = c$ **then**
14: Program $H_3(K'_{\hat{w}}, i) \leftarrow \mathtt{sk}[t_i] - \overline{bs}$
15: **else**
16: Program $H_3(K'_{\hat{w}}, i) \leftarrow \mathtt{sk}[t_i]$
17: **end if**
18: **end for**
19: Send $(K_{\hat{w}}, ST_c, c)$ to the server.

□

Corollary 1. *(Adaptive forward privacy of $\mathtt{FB\text{-}DSSE}$). $\mathtt{FB\text{-}DSSE}$ is forward-private.*

From **Theorem 3**, we can infer that $\mathtt{FB\text{-}DSSE}$ achieves forward privacy, since the leakage function $\mathcal{L}^{Update}$ of $\mathtt{FB\text{-}DSSE}$ does not leak the keyword during update as defined in **Definition 5**.

Corollary 2. *(Adaptive Type-I$^-$ backward privacy of $\mathtt{FB\text{-}DSSE}$). $\mathtt{FB\text{-}DSSE}$ is Type-I$^-$ backward-private.*

From **Theorem 3**, we can infer that $\mathtt{FB\text{-}DSSE}$ achieves Type-I$^-$ backward privacy, since the leakage functions of $\mathtt{FB\text{-}DSSE}$ leak less information than the leakage functions in **Definition 6**.

Remark. For the multi-block extension MB−FB−DSSE, the underlying construction is almost same as FB−DSSE except that it encrypts multi-block bit string **bs** rather than one bit string bs. Hence, it inherits the forward privacy and Type-I$^-$ backward privacy of FB−DSSE.

4.6 Experimental Analysis

Our schemes deploy simple symmetric primitives to achieve strong backward privacy, which are more efficient than the schemes from [21, 55] because the authors of [21, 55] deploy ORAM [52, 53] to achieve strong backward privacy. The scheme Janus++ from [56] is the most efficient backward-private scheme which is based on the scheme Janus from [21]. However, Janus++ only achieves Type-III backward privacy. Table 4.3 compares the results.

Table 4.3: Comparison of computing overhead

Scheme	Search	Update
Janus [21]	$O(n_w + d_w) \cdot t_{\text{PE.Dec}}$	$O(1) \cdot (t_{\text{PE.Enc}}$ or $t_{\text{PE.Punc}})$
Janus++ [56]	$O(n_w + d_w) \cdot t_{\text{SPE.Dec}}$	$O(1) \cdot (t_{\text{SPE.Enc}}$ or $t_{\text{SPE.Punc}})$
MONETA [21]	$\hat{O}(a_w logN + log^3 N) \cdot t_{\text{SKE}}$	$\hat{O}(log^2 N) \cdot t_{\text{SKE}}$
ORION [55]	$O(n_w log^2 N) \cdot t_{\text{SKE}}$	$O(log^2 N) \cdot t_{\text{SKE}}$
FB−DSSE [Sec. 4.4.2]	$O(a_w) \cdot t_{ma}$	$O(1) \cdot t_{ma}$
MB−FB−DSSE [Sec. 4.4.3]	$O(a_w) \cdot t_{ma} \cdot B$	$O(1) \cdot t_{ma} \cdot B$

N is the number of keyword/file-identifier pairs, n_w is the number of files currently matching keyword w, d_w is the number of deleted entries for keyword w, and a_w is the total number of updates corresponding to keyword w. $t_{\text{PE.Enc}}$, $t_{\text{PE.Dec}}$ and $t_{\text{PE.Punc}}$ are the encryption, decryption and puncture time of a public puncturable encryption. $t_{\text{SPE.Enc}}$, $t_{\text{SPE.Dec}}$ and $t_{\text{SPE.Punc}}$ are the encryption, decryption and puncture time of a symmetric puncturable encryption. t_{SKE} is the encryption and decryption time of a symmetric key encryption. t_{ma} is the computational time of a modular addition, and B is the number of blocks. $\hat{O}$ notation hides polylogarithmic factors.

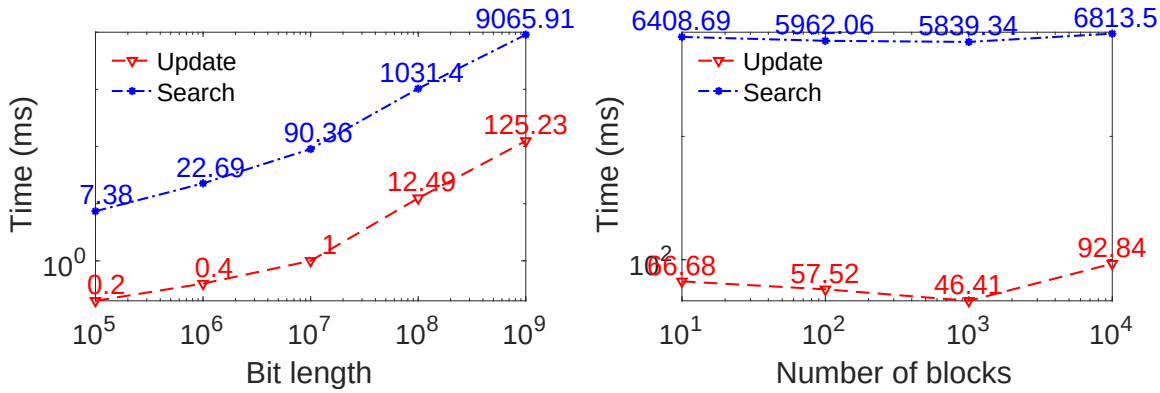
Now we are ready to give the experimental evaluation. We evaluate the performance of our schemes in a testbed of one workstation. This machine plays the roles of the client and server. The hardware and software of this machine are as follows: Mac Book Pro, Intel Core i7 CPU @ 2.8GHz RAM 16GB, Java Programming Language, and macOS 10.13.2. Note that we use the bitmap index to denote file identifiers, and we tested the search and update time for one keyword. We use the "BigInteger" with different bit lengths to denote the bitmap index with different sizes, which acts as the database with a different number of files. The relation between the i-th bit and the actual file is out of our scope.

For every keyword, we run the update operation **Update** for this keyword 20 times. In other words, every keyword has 20 entries. The update time includes the client token generation time and server update time, and the search time includes the token generation time, the server search time, and the client decryption time. Note that the result only depends on the maximum number of files supported by the system (the bit length), but not the actual number of files in the server.

First, we give the search and update time of FB−DSSE with different bit length in Fig. 4.2a. The bit length refers to ℓ, which is equal to the maximum number of files supported by the system. From Fig. 4.2a, we can see that the update and search time grow with the increasing of bit length. We also can observe that the update time with the bit length from 10^5 to 10^6 does not increase a lot. This is because the addition and modular have not contributed too much when the bit length is less than 10^6.

In Fig. 4.2b, we evaluate the search and update time of MB−FB−DSSE with different number of blocks, where the total bit length is 10^9. When we divide one bit string (10^9) into different blocks, we can see that the running time is lesser than one block in Fig. 4.2a. For the number of blocks from 10 to 10^3, it can be seen that the update and search time decrease. However, when the number of blocks is 10^4, the update and search time increase due to the fact that when the bit string decreases to a certain length, the addition and modular time do not decrease too much.

To support an extreme large number of files (such as 1 billion), MB−FB−DSSE may be preferred than FB−DSSE. For example, the search and update time of MB−FB−DSSE are 5.84s and 46.41ms, respectively, where the number of blocks is 10^3, and the bit length of each block is 10^6. However, the search and update time of FB−DSSE supporting 1 billion files (bit length = 10^9) are 9.07s and 125.23ms, respectively.



(a) The running time of FB−DSSE with different bit length

(b) The running time of MB−FB−DSSE with different number of blocks for 10^9 bits

Figure 4.2: The running time of our schemes

4.7 Conclusion

In this chapter, we gave a DSSE scheme with stronger (named Type-I$^-$) backward privacy, which also achieves forward privacy efficiently. Moreover, to make it scalable for supporting billions of files with high efficiency, we extended our first scheme to the multi-block setting. From the experimental analysis, we can see that the efficiency of the first scheme with an extremely large bit length can be improved by splitting a long bit string into multiple short bit strings.

Chapter 5

Forward and Backward Private DSSE for Range Queries

Due to its capabilities of searches and updates over the encrypted database, the dynamic searchable symmetric encryption (DSSE) has received considerable attention recently. To resist leakage abuse attacks, a secure DSSE scheme usually requires forward and backward privacy. However, the existing forward and backward private DSSE schemes either only support single keyword queries or require more interactions between the client and the server.

In this chapter, we first give a new leakage function for range queries, which is more complicated than the one for single keyword queries. Furthermore, we propose a concrete forward and backward private DSSE scheme by using a refined binary tree data structure. Finally, the detailed security analysis and extensive experiments demonstrate that our proposal is secure and efficient, respectively.

5.1 Introduction

Outsourcing data to the cloud is a cost-effective and reliable way to store large amounts of data. However, at the same time, it exposes data to a server that is not always trusted. Hence, the security and privacy of outsourced data should be addressed before using cloud storage. A simple method to mitigate these problems is to encrypt data before outsourcing. Unfortunately, encryption reduces the usability, especially the searchability, of the data due to the nature of encryption. To solve this problem, searchable symmetric encryption (SSE) has been introduced [10, 11]. It encrypts the data while preserving the searchability of the data. Compared with other techniques for enabling searchability over ciphertexts [52, 61], the clear advantages of SSE is its efficiency.

Traditional SSE schemes cannot support updates over an encrypted database. This substantially limits its applications. To support updates of encrypted databases, dynamic SSE (DSSE) has been proposed in [15, 17]. However, updates leak information about data (see [16]). Zhang et al. [18] demonstrated file-injection attacks that break the privacy of client queries by injecting a small number of files into an encrypted database. To deal with the attacks, forward and backward privacy notions have been introduced informally in [19]. Later, the notions have been formalized in [20] and [21], respectively. In particular, Bost et al. [21] defined three different levels of backward privacy, namely, Type-I, Type-II and Type-III, where Type-I is the most secure and Type-III is the least secure. Many other forward and backward private DSSE schemes have also been proposed (see [21, 56] for instance). Unfortunately, a majority of published forward and backward private DSSE schemes support

single keyword queries only. This greatly reduces their useability. In many applications, we need more expressive search queries, such as range queries, for instance.

Consider range queries. In a naïve solution, one could query all possible values in a range. This solution is not efficient if the range is large, as it requires a large communication overhead. To process range queries more efficiently and reduce communication cost, Faber et al. [13] applied a binary tree to the OXT scheme of Cash et al. [12]. Their solution works for static databases only and does support updates. Zuo et al. [54] designed two DSSE schemes using a new binary tree data structure. Their schemes support both range queries and updates. Their first scheme (`SchemeA`) based on the framework of [20] achieves forward privacy. However, it inherits the low efficiency of the scheme from [20] due to the application of computationally expensive public-key cryptographic operations. For the second scheme (`SchemeB`), the authors combined the bit string representation with the Paillier encryption [22]. The second scheme achieves backward privacy. The maximum number of files the scheme can support is equal to the length of the message space for the Paillier encryption. For a typical implementation, the message length is very small (around 1024 bits), and therefore a scheme can support a limited number of files. To reduce storage requirements, the authors homomorphically add the ciphertexts, and consequently, their scheme loses forward privacy. In addition, they did not provide a detailed backward privacy analysis. Later, Wang et al. [62] suggested a generic forward private DSSE with range queries by adapting the `ShemeA` from [54]. To achieve backward privacy, they extended their scheme by applying the generic backward private construction of [21]. Unfortunately, to support the backward privacy, their scheme requires another roundtrip between the client and the server. In other words, the client needs to re-encrypt the matched files and send them back to the server, which is not efficient.

Recently, Zuo et al. [63] designed an efficient DSSE scheme with forward and stronger backward privacy by combining the bitmap index with simple symmetric encryption with homomorphic addition. To support very large databases, they extended their first scheme to the multiple block setting. However, their schemes support single keyword queries only.

Our Contributions. We develop an efficient forward and backward private DSSE scheme that supports range queries by extending the scheme from the work [63]. The scheme is further called `FBDSSE-RQ`. It requires one roundtrip only. The comparison with previous works is given in Table 5.1. The list given below details our contributions.

- First, we refine the construction of the binary tree from [54]. For our binary tree, we label all nodes by keywords. Names of nodes are derived from their leaf nodes rather than from the order of node insertion (see Section 5.2.2 for more details). We also modify algorithms for the binary tree.
- We define a new backward privacy for our range queries named Type-R. Compared with single keyword queries, range queries introduce more leakages. We map a range

Table 5.1: Comparison to previous works

Scheme	Forward Privacy	Backward Privacy	Range Queries	Number of Roundtrips
FIDES [21]	✓	Type-II	✗	2
DIANA_{del} [21]	✓	Type-III	✗	2
Janus [21]	✓	Type-III	✗	1
Janus++ [56]	✓	Type-III	✗	1
MONETA [21]	✓	Type-I	✗	3
FB-DSSE [63]	✓	Type-I$^-$	✗	1
SchemeA [54]	✓	✗	✓	1
SchemeB [54]	✗	Unknown	✓	1
Generic [62]	✓	✗	✓	1
Extension [62]	✓	Type-II	✓	2
Our scheme	✓	Type-R	✓	1

query into several keywords that are assigned to nodes of our binary tree. For a range search query $[a, b]$, a query leaks the number of keywords, the total number of updates and update time for each keyword, the repetition of these keywords, and the final results for the range query[1], and for the update with value v, it leaks the number of keywords that have been updated (the number of levels of the binary tree). See Sections 5.4 and 5.5 for details.

- We describe a forward and Type-R backward private DSSE for range queries. The scheme called FBDSSE-RQ uses our refined binary tree and is based on the FB-DSSE from [63]. In addition, it only requires one roundtrip. Our scheme is more efficient than the extension scheme from [62]. For every search, the scheme from [62] needs to re-encrypt the search results and send them back to the server, which incurs high computational and communication costs. See Section 5.5 for details.

- Finally, the security analysis and implementation experiments demonstrate that the scheme achieves claimed security goals and is practical.

5.1.1 Related Work

For the completeness and consistency of this Chapter, we list some related works that already appeared in Section 2.1.1, 3.1.1 and 4.1.1. Searchable symmetric encryption (SSE) was introduced by Song et al. [10]. In their scheme, a client encrypts every keyword of a file. For a search query, the client first encrypts a keyword and then finds a match by comparing the (encrypted) keyword to (encrypted) keywords of all the files. As a result, the search time is linear with the number of file/keyword pairs. To reduce the search time, Curtmola et al. [11] deployed an inverted index data structure. Consequently, their SSE scheme obtains

[1] If $a = b$, the leakage of the search query would be same as Type-I$^-$. Moreover, Type-R does not leak the insertion time of each file identifier while Type-II does.

sublinear search time. In [11], the authors formally defined the SSE security model. There is a large number of followup papers studying different aspects of SSE. For instance, SSE with expressive queries is examined in [12, 13, 54], SSE for multi-client setting is explored in [11, 37], dynamic SSE – in [15, 17] and locality SSE – in [64, 65].

Once a database is encrypted, SSE schemes do not allow for updating the encrypted database. To support updates of the encrypted database, dynamic SSE (DSSE) schemes have been introduced in [15, 17]. Early DSSE schemes are, however, vulnerable to file-injection attacks [16, 18]. To deal with the attacks, forward and backward privacy have been informally introduced in [19]. Bost [20] formalized the forward privacy. Later Bost et al. [21] defined three levels of backward privacy (Type-I to Type-III, ordered from the most to the least secure). Sun et al. [56] designed a DSSE called `Janas++`, which achieves Type-III backward privacy by replacing (public-key) puncturable encryption (PE) with symmetric puncturable encryption (SPE).

A majority of forward and/or backward private DSSE schemes support single keyword queries only. Faber et al. [13] constructed an SSE scheme that accepts range queries. The scheme applies a binary tree data structure to the OXT scheme of Cash et al. [12]. However, the scheme is static (does not allow updates). To design a DSSE for range queries, Zuo et al. [54] deployed a new binary tree data structure. They described two solutions. The first one is based on the scheme by Bost [20]. It achieves forward privacy. The second solution applies the Paillier cryptosystem [22], and it is backward private. Unfortunately, the solution can support a limited number of files only. This weakness is due to a limited length of the message space of the Paillier cryptosystem. Wang et al. [62] designed a generic forward private DSSE for range queries. The generic construction applies the framework of `SchemeA` from [54]. They also extended their first scheme by integrating the generic backward private construction from [21]. The scheme achieves Type-II backward privacy. The scheme, however, requires two roundtrips between the client and the server, which is not efficient. Independently, Demertzis et al. [66] developed several SSE schemes for range queries with different security and efficiency tradeoffs by using the binary tree data structure. To support updates, they deploy several independent SSE instances and periodically consolidate them. As far as information leakage is concerned, the schemes leak not only the number of keywords queried but also the level of each keyword in the binary tree.

Recently, Zuo et al. [63] introduced a forward and stronger backward private DSSE, which requires one roundtrip only. Moreover, they introduced a new notion of backward privacy (named Type-I^{-}). Compared to Type-I [21], Type-I^{-} does not leak the insertion time of matching files. This is achieved by deploying a bitmap index and simple symmetric encryption with homomorphic addition. Experiments show that the DSSE scheme is efficient and practical. Nevertheless, it can support single keyword queries only. To the best of our knowledge, there is no forward and backward private DSSE that can process range queries with one roundtrip only.

There is also another line of investigation that explores the usage of trusted hardware (SGX) in order to obtain secure DSSE (see [57, 67], for example). In this chapter, we focus on constructing a secure DSSE without a trusted third party. The readers, who are interested in this aspect of DSSE design, are referred to [57, 67].

5.1.2 Organization

The remaining sections are organized as follows. In Section 5.2, we give the necessary background information and preliminaries. In Section 5.3, we define our DSSE model. The forward and backward privacy notions for our range queries are given in Section 5.4. In Section 5.5, we give our forward and backward private DSSE for range queries. The security analysis is given in Section 5.6. Section 5.7 discusses implementation of our scheme and its efficiency. Finally, Section 5.8 concludes the chapter.

5.2 Preliminaries

let λ be the security parameter. We use a bitmap index to represent file identifiers in the same way as in [63]. For a database with y files, we set a bit string bs of length y. If there exists file f_i, we set the i-th bit of bs to 1. Otherwise, it is set to 0. Fig. 5.1 illustrates setup, addition and deletion of file identifiers. In particular, Fig. 5.1(a) shows a bitmap index for a database that can store up to $y = 5$ files. The index tells us that the database contains a single file f_2. Fig. 4.1(b) illustrates addition of file f_1 to the database, i.e. the bit string 00010 (that corresponds to f_1) is added to the index. Fig. 4.1(c) displays operations on the index, when the file f_2 is deleted from the database. This can be done either by subtracting the string 00100 from the index or by adding $-(00100)_2 = (11100)_2$ to the index (note that operations are performed modulo 2^5).

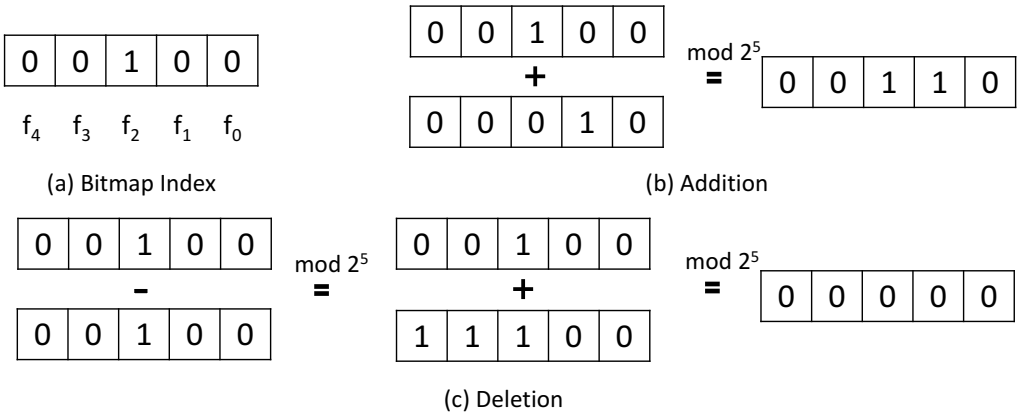


Figure 5.1: Illustration of bitmap index operations

5.2.1 Simple Symmetric Encryption with Homomorphic Addition

Following [60], a simple symmetric encryption with homomorphic addition $\Pi = (\texttt{Setup}, \texttt{Enc}, \texttt{Dec}, \texttt{Add})$ is described by following four algorithms:

- $n \leftarrow \texttt{Setup}(1^\lambda)$: For a security parameter λ, it outputs a public parameter n, where $n = 2^y$ and y is the maximum number of files a scheme can support.
- $c \leftarrow \texttt{Enc}(sk, m, n)$: For a message m ($0 \leq m < n$), a public parameter n and a random secret key sk ($0 \leq sk < n$), it computes a ciphertext $c = sk + m \bmod n$. For every encryption, the secret key sk needs to be stored, and it can be used once only.
- $m \leftarrow \texttt{Dec}(sk, c, n)$: For a ciphertext c, a public parameter n and a secret key sk, it recovers the message $m = c - sk \bmod n$.
- $\hat{c} \leftarrow \texttt{Add}(c_0, c_1, n)$: For two ciphertexts c_0, c_1 and a public parameter n, it computes $\hat{c} = c_0 + c_1 \bmod n$, where $c_0 \leftarrow \texttt{Enc}(sk_0, m_0, n)$, $c_1 \leftarrow \texttt{Enc}(sk_1, m_1, n)$, $n \leftarrow \texttt{Setup}(1^\lambda)$ and $0 \leq sk_0, sk_1 < n$.

We claim that the above defined encryption supports homomorphic addition, in the sense that, knowing two ciphertexts $c_0 = m_0 + sk_0 \bmod n$ and $c_1 = m_1 + sk_1 \bmod b$, anybody can create $\hat{c} = c_0 + c_1 \bmod n$. However, to decrypt $\hat{c}$ and recover $m_0 + m_1 \bmod n$, one needs to know $sk_0 + sk_1 \bmod n$. To prove validity of the claim, it is enough to check that

$$\texttt{Dec}(\hat{sk}, \hat{c}, n) = \hat{c} - \hat{sk} \bmod n = m_0 + m_1 \bmod n,$$

where $\hat{sk} = sk_0 + sk_1 \bmod n$.

Note that Π enjoys perfect security as long as secret keys are used once only. To see that this is true is enough to note that our encryption becomes the well-known one-time pad (OTP) when the secret key is chosen randomly and uniformly for each new message.

Perfect Security [60]. We say Π is perfectly secure if for any adversary $\mathcal{A}$, its advantage is negligible or

$$\mathbf{Adv}^{\texttt{PS}}_{\Pi,\mathcal{A}}(\lambda) = |\Pr[\mathcal{A}(\texttt{Enc}(sk, m_0, n)) = 1] -$$
$$\Pr[\mathcal{A}(\texttt{Enc}(sk, m_1, n)) = 1]| \leq \epsilon,$$

where $n \leftarrow \texttt{Setup}(1^\lambda)$, the secret key sk ($0 \leq sk < n$) is kept secret and $\mathcal{A}$ chooses m_0, m_1 s.t. $0 \leq m_0, m_1 < n$.

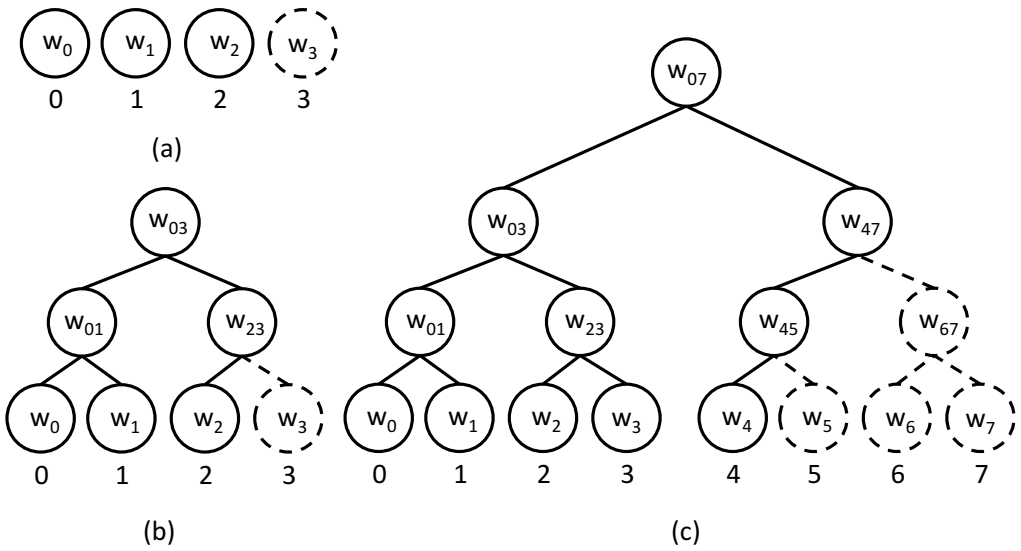


Figure 5.2: Binary Tree

Algorithm 11 Binary Tree

TGen(d)

if $d \leq 0$ **then**
  return $\perp$
else if $d = 1$ **then**
  Generate one node n and set this node as BT.
  Associate value 0 to this node and name it as w_0.
  return BT
else
  Generate 2^ℓ leaf nodes ▷ $2^{\ell-1} < d \leq 2^\ell$
  Associate each leaf node with each value $v \in 2^\ell$ and name the corresponding leaf node as w_v.
  for $i = \ell - 1$ to 0 **do**
    Generate 2^i nodes.
    for each node **do**
      Set its left and right child to two consecutive nodes from the previous level, where the value of its $leftmost$ is even and the value of its $rightmost$ is odd.
      Name this node as w_{ab}, where a and b are the values associated with its $leftmost$ and $rightmost$.
    end for
  end for
  Set the root node as BT
  return BT
end if

TGetCover(q, BT) ▷ $q = [a, b]$, where $0 \leq a < b < d$

BRC, Temp, Parent ← Empty Set
for $i = a$ to b **do**
  Temp ← Temp $\cup\, w_i$ ▷ Put all the leaf nodes to the temp set Temp.
end for
while Temp $\neq \perp$ **do**
  for two nodes in Temp have the same parent **do**
    Remove these two nodes from Temp, and put the parent node to the set Parent.
  end for
  Move the remaining nodes from Temp to BRC.
  Temp ← Parent, Parent ← $\perp$
end while
return BRC

TPath(v, BT)

PT ← Empty Set
$w \leftarrow w_v$
while $w \neq \perp$ **do**
  PT ← PT $\cup\, w$
  $w \leftarrow w \cdot parent$
end while
return PT

5.2.2 Binary Tree

We revisit the binary tree BT from [54]. For simplicity, we always use a perfect (a.k.a. full) binary tree and denote its root $root$ as BT. A perfect binary tree is a binary tree with 2^ℓ leaf nodes, where $\ell+1$ is the number of levels. The root exists at the level 0 and leaves belong to the level ℓ. For range queries on attribute A (e.g. age) with range $R = \{0, 1, \cdots, d-1\}$, each leaf of BT is associated with a value v from R. For example, in Fig. 5.2(a), d is 3. To form a perfect binary tree, we need to add an additional leaf (the dot-line node in Fig. 5.2(a)). For Fig. 5.2(c), d is 5. Every node in BT has three pointers, which are initially set to null. The three pointers are $parent$, $left$ and $right$. The $parent$ links the node with its parent. The pointers $left$ and $right$ connect the node with its left and right children, respectively. We also define the $leftmost$ child and $righmost$ child. The $leftmost$ leaf is a node, which is the left child of its parent and all parents are left children of their ancestors. The $rightmost$ leaf is defined similarly but for right child. For example, in Fig. 5.2(b), w_0 is the $leftmost$ leaf of w_{03} and w_3 is the $rightmost$ leaf of w_{03}. Now, we are ready to describe a collection of algorithms for BT (see Algorithm 11 for precise definition).

- $\mathtt{BT} \leftarrow \mathtt{TGen}(d)$: It takes d and outputs a perfect binary tree BT for 2^ℓ leaf nodes, where $2^{\ell-1} < d \leq 2^\ell$ and ℓ is the smallest such integer. For example, Fig. 5.2(b) and Fig. 5.2(c) illustrate a tree constructed for $d = 3$ and $d = 5$ leaves, respectively.
- $\mathtt{BRC} \leftarrow \mathtt{TGetCover}(q, \mathtt{BT})$: The algorithm takes a range $q = [a, b]$ and a binary tree BT as its input and outputs the best range cover BRC that contains all leaves in the range $[a, b]$, where $0 \leq a < b < d$[2]. Note that a BRC has to include the smallest number of parent nodes of leaves in the range. Consider the tree depicted in Fig. 5.2(c), $\mathtt{BRC} = \{w_{23}, w_4\}$ for range query $q = [2, 4]$.
- $\mathtt{PT} \leftarrow \mathtt{TPath}(v, \mathtt{BT})$: The algorithm takes a value v and a binary tree BT as its input and outputs a set PT of nodes that belong to the path traversing from the leaf w_v to the $root$, where $0 \leq v < d$. For instance, consider the tree in Fig. 5.2(c). For $v = 1$ (or the leaf w_1), the set $\mathtt{PT} = \{w_1, w_{01}, w_{03}, w_{07}\}$.

5.2.3 Notations

Notations used in the chapter are given in Table 5.2.

5.3 DSSE definition and Security Model

For range queries, we assume that each file f is characterised by an attribute A (e.g. age), whose value v belongs to the range $R = \{0, 1, \cdots, d-1\}$. We assign the range values

[2] If $a = b$, it becomes a single keyword query for keyword w_a.

Table 5.2: Notations

v	The value in a range query
BT	The full binary tree
$\ell + 1$	The number of levels of the binary tree, where the root is in level 0 and the leaves are in level ℓ
d	The boundary of our range query
R	The set of values for our range query $\{0, 1, \cdots, d-1\}$
$[a, b]$	A range query
BRC	The set of least number of nodes to cover range $[a, b]$
PT	The set of nodes in the path from a leaf to the root
DB	A database
λ	The security parameter
ST_c	The current search token for a keyword w
EDB	The encrypted database EDB which is a map
F	A secure PRF
W	The set of all keywords of the database DB
CT	A map stores the current search token ST_c and counter c for every keyword in **W**
f_i	The i-th file
bs	The bit string which is used to represent the existence of files
y	The length of bs
e	The encrypted bit string
Sum_e	The sum of the encrypted bit strings
sk	The one time secret key
Sum_{sk}	The sum of the one time secret keys

to the leaves of our binary tree BT, as shown in Fig. 5.2(b). Consequently, each file contains not only the keyword of its leaf but also the keywords associated with its ancestors.

A database DB stores a list of file-identifier/keyword-set pairs or DB$= (f_i, \mathbf{W}_i)_{i=1}^{y}$, where $f_i \in \{0,1\}^{\lambda}$ is the file identifier, $\mathbf{W}_i$ is the keyword set and y is the total number of files in DB. For example, consider the tree from Fig. 5.2(b), the file f_0 is associated with the range value 0 and contains keywords from the set $\mathbf{W}_0 = \{w_0, w_{01}, w_{03}\}$. We denote the collection of all distinct keywords in DB by $\mathbf{W} = \cup_{i=1}^{y} \mathbf{W}_i$. The notation $|\mathbf{W}|$ means the total number of keywords in the set **W** (or cardinality of the set). The total number of file-identifier/keyword pairs is denoted by $N = \sum_{i=1}^{y} |\mathbf{W}_i|$.

A set of files that satisfy a range query q is denoted by DB(q). Note that we use the bitmap index to represent the file identifiers. For a search query q, the result is a bit string bs, which represents a list of file identifiers in DB(q). For an update query u, a bit string bs is used to update a list of file identifiers. Moreover, we isolate the actual files from the metadata (e.g., file identifiers). We focus on the search of the metadata only. We ignore the retrieval process of encrypted files from the database.

5.3.1 DSSE Definition

A DSSE scheme consists of an algorithm **Setup** and two protocols **Search** and **Update** that are executed between a client and a server. They are described as follows:

- ($\texttt{EDB}$, σ) $\leftarrow$ **Setup**(1^λ, $\texttt{DB}$): For a security parameter λ and a database $\texttt{DB}$, the algorithm outputs a pair: an encrypted database $\texttt{EDB}$ and a state σ. $\texttt{EDB}$ is stored by the server and σ is kept by the client.
- ($\mathcal{I}$, $\perp$) $\leftarrow$ **Search**(q, σ; $\texttt{EDB}$): For a state σ, the client issues a query q and interacts with the server who holds $\texttt{EDB}$. At the end of the protocol, the client outputs a set of file identifiers $\mathcal{I}$ that match q and the server outputs nothing.
- (σ', $\texttt{EDB}'$) $\leftarrow$ **Update**(σ, op, in; $\texttt{EDB}$): For a state σ, the operation $op \in \{add, del\}$ and a collection of $in = (f, \mathbf{w})$ pairs, the client requests the server (who holds $\texttt{EDB}$) to update database by adding/deleting files specified by the collection in. Finally, the protocol returns an updated state σ' to the client and an updated encrypted database $\texttt{EDB}'$ to the server.

Remark. In literature, there are two result models for SSE schemes. In the first one (considered in the work [12]), the server returns encrypted file identifiers $\mathcal{I}$, so the client needs to decrypt them. In the second one (studied in the work [20]), the server returns the file identifiers to the client as plaintext. In our work, we consider the first variant, where the protocol returns encrypted file identifiers.

5.3.2 Security Model

DSSE security is modeled by the Real and Ideal worlds called $\texttt{DSSEREAL}$ and $\texttt{DSSEIDEAL}$, respectively. The behavior of $\texttt{DSSEREAL}$ is exactly the same as the original DSSE. However, $\texttt{DSSEIDEAL}$ reflects a behavior of a simulator $\mathcal{S}$, which takes the leakages of the original DSSE as input. The leakages are defined by the function $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Search}, \mathcal{L}^{Update})$, which details what information the adversary $\mathcal{A}$ can learn during execution of the **Setup** algorithm, **Search** and **Update** protocols.

If the adversary $\mathcal{A}$ can distinguish $\texttt{DSSEREAL}$ from $\texttt{DSSEIDEAL}$ with a negligible advantage, we can say that leakage of information is restricted to the leakage $\mathcal{L}$. More formally, we consider the following security game. The adversary $\mathcal{A}$ interacts with one of the two worlds $\texttt{DSSEREAL}$ or $\texttt{DSSEIDEAL}$ which are described as follows:

- $\texttt{DSSEREAL}_{\mathcal{A}}(\lambda)$: On input a database $\texttt{DB}$, which is chosen by the adversary $\mathcal{A}$, it outputs $\texttt{EDB}$ to $\mathcal{A}$ by running **Setup**(λ, $\texttt{DB}$). $\mathcal{A}$ performs search queries q (or update queries (op, in)). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.

- $\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda)$: Simulator $\mathcal{S}$ outputs the simulated $\texttt{EDB}$ with the input $\mathcal{L}^{Setup}(\lambda, \texttt{DB})$). For search queries q (or update queries (op, in)) generated by the adversary $\mathcal{A}$, the simulator $\mathcal{S}$ replies by using the leakage function $\mathcal{L}^{Search}(q)$ (or $\mathcal{L}^{Update}(op, in)$). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.

Definition 7. *Given a DSSE scheme and the security game described above. The scheme is $\mathcal{L}$-adaptively-secure if for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ (with the input $\mathcal{L}$) such that,*

$$|\Pr[\texttt{DSSEREAL}_{\mathcal{A}}(\lambda) = 1] - \Pr[\texttt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]|$$

$$\leq negl(\lambda).$$

Leakage Function. Before defining the leakage function, we define a range query $q = (t, [a, b]) = \{t, w\}_{w \in \texttt{BRC}}$, where $\texttt{BRC}$ is the best range cover of range $[a, b]$. An update query $u = (t, op, (v, bs)) = \{t, op, (w, bs)\}_{w \in \texttt{PT}(v)}$, where t is the timestamp, $\texttt{PT}$ contains all keywords in the path from the leaf node of v to the root, op is the update operation and bs denotes a list of file identifiers to be updated. For a list of search queries Q, we define a search pattern $\texttt{sp}(q) = \{t : (t, w)\}_{w \in \texttt{BRC}}$, where t is a timestamp and $q \in Q$. The search pattern leaks the repetition of search queries on q. Denote a result pattern $\texttt{rp}(q) = \overline{bs}$, where $\overline{bs}$ represents all file identifiers that match the range query q. Note that, after a search query, we implicitly assume that the server knows the final result $\overline{bs}$, since the client may retrieve the file identifiers represented by $\overline{bs}$ which is not described in this chapter. Moreover, the server can infer if a range query contains other range queries or not by looking at $\overline{bs}$.

5.4 Forward and Backward Privacy for Our Range Queries

To support range queries, we incorporate the binary tree data structure (see Section 5.2.2 for details). For an update with a value v, we need to update every node (keyword) in the path from the corresponding leaf node to the root node, where the value v is within the boundaries of the current binary tree. For the update with a value v, we need to issue several updates (all keywords from the leaf to the root). Hence the number of updates (the number of levels of the binary tree) is leaked.

5.4.1 Forward Privacy

Informally, for any adversary who may continuously observe the interactions between the server and the client, forward privacy guarantees that an update does not leak information about the newly added files that match the previously issued queries. The definition given below is taken from [20]:

Definition 8. *A $\mathcal{L}$-adaptively-secure DSSE scheme is forward-private if the update leakage function $\mathcal{L}^{Update}$ can be written as*

$$\mathcal{L}^{Update}(op, in) = \mathcal{L}'(op, \{(f_i, \mu_i)\}),$$

where f_i is the identifier of the modified file, μ_i is the number of keywords corresponding to the updated file f_i.

Remark. For our range query, the leakage function will be $\mathcal{L}^{Update}(op, v, bs) = \mathcal{L}'(op, bs, \ell + 1)$, where $\ell + 1$ is the number of levels of the full binary tree $\mathtt{BT}$.

5.4.2 Backward Privacy

Given a time interval in which two search queries for the same range occur. Backward privacy ensures that there is no leak of information about the files that have been previously added and later deleted. Note that information about files leak if the second search query is issued after the files are added but before they are deleted. In [63], Zuo et al. formulated a stronger level of backward privacy named Type-I$^-$ for single keyword queries. To deal with range queries, we map a range query to several keywords. For our range queries, to update a value, we need to update every keyword, which contains this value. Hence the update leaks the number of keywords corresponding to the value, which is the number of levels of the binary tree $\ell + 1$. This type of backward privacy is called Type-R.

- Type-R: Given a time interval between two calls issued for a range query q. Then it leaks the files that currently match q, and the total number of updates and the time of each update for each w, where $w \in \mathtt{BRC}$. The update of a leaf (value v) leaks the number of keywords corresponding to the value.

To define Type-R formally, we need to introduce $\mathtt{Time}$. For a range query q, $\mathtt{Time}(q)$ lists the timestamp t of all updates corresponding to each w, where $w \in \mathtt{BRC}$. Formally, for a sequence of update queries Q':

$$\mathtt{Time}(q) = \{t : (t, op, (w, bs))\}_{w \in \mathtt{BRC}}.$$

Definition 9. *A $\mathcal{L}$-adaptively-secure DSSE scheme is Type-R backward-private iff the search and update leakage function $\mathcal{L}^{Search}$, $\mathcal{L}^{Update}$ can be written as:*

$$\mathcal{L}^{Update}(op, v, bs) = \mathcal{L}'(op, \ell + 1),$$

$$\mathcal{L}^{Search}(q) = \mathcal{L}''(\mathtt{sp}(q), \mathtt{rp}(q), \mathtt{Time}(q)),$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless, $\ell + 1$ is the number of levels of the full binary tree $\mathtt{BT}$.

5.5 Forward and Backward Private DSSE for Range Queries

Now, we are ready to give our forward and backward private DSSE for range queries. We call it FBDSSE−RQ and it is defined by Algorithm 12. Our DSSE is based on the framework of [63], a simple symmetric encryption with homomorphic addition $\Pi = (\texttt{Setup}, \texttt{Enc}, \texttt{Dec}, \texttt{Add})$, and a keyed PRF F_K with key K. The scheme is defined by the following algorithm and two protocols:

Algorithm 12 FBDSSE−RQ

Setup(1^λ)

Client:

$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
CT, EDB $\leftarrow$ empty map
Set the range boundary d.
return $(\texttt{EDB}, \sigma = (n, d, K, \textbf{CT}))$

Update$(v, bs, \sigma; \texttt{EDB})$ ▷ $0 \leq v < d$

Client:

$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{PT} \leftarrow \texttt{TPath}(v, \texttt{BT})$
for $w \in \texttt{PT}$ **do**
  $K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    $c \leftarrow -1, ST_c \leftarrow \{0,1\}^\lambda$
  end if
  $ST_{c+1} \leftarrow \{0,1\}^\lambda$
  $\textbf{CT}[w] \leftarrow (ST_{c+1}, c+1)$
  $UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
  $C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
  $sk_{c+1} \leftarrow H_3(K'_w, c+1)$
  $e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
  Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to the server.
end for

Server:

Upon receiving $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$
Set $\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(q, \sigma, \texttt{EDB})$ ▷ $q = [a, b]$, where $0 \leq a < b < d$.

Client:

$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{BRC} \leftarrow \texttt{TGetCover}(q, \texttt{BT})$
for $w \in \texttt{BRC}$ **do**
  $K_w || K'_w \leftarrow F_K(w)$, $(ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    return $\perp$
  end if
end for
Send $\{(K_w, ST_c, c)\}_{w \in \texttt{BRC}}$ to the server.

Server:

$Sum \leftarrow 0$
for each (K_w, ST_c, c) **do**
  $Sum_e \leftarrow 0$
  for $i = c$ to 0 **do**
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $(e_i, C_{ST_{i-1}}) \leftarrow \texttt{EDB}[UT_i]$
    $Sum_e \leftarrow \texttt{Add}(Sum_e, e_i, n)$
    Remove $\texttt{EDB}[UT_i]$
    if $C_{ST_{i-1}} = \perp$ **then**
      $Break$
    end if
    $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
  end for
  $\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
  $Sum \leftarrow \texttt{Add}(Sum, Sum_e, n)$
end for
Send Sum to the client.

Client:

$Sum_{sk} \leftarrow 0$
for $w \in \texttt{BRC}$ **do**
  for $i = c$ to 0 **do**
    $sk_i \leftarrow H_3(K'_w, i)$
    $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
  end for
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum, n)$
return bs

- $(\texttt{EDB}, \sigma = (n, d, K, \textbf{CT})) \leftarrow$ **Setup**(1^λ): The algorithm is run by a client. It takes the security parameter λ as input. Then it chooses a secret key K and an integer n,

where $n = 2^y$ and y is the maximum number of files that this scheme can support. Moreover, it sets the range query boundary d, two empty maps $\texttt{EDB}$ and **CT**, where $R = \{0, \cdots, d-1\}$ is set of values for our range queries and the two maps are used to store the encrypted database as well as the current search token ST_c and the current counter c (the number of updates) for each keyword $w \in \mathbf{W}$, respectively. Finally, it outputs encrypted database $\texttt{EDB}$ and the state $\sigma = (n, d, K, \mathbf{CT})$. The client keeps $(d, K, \mathbf{CT})$ secret.

- $(\sigma', \texttt{EDB}') \leftarrow$ **Update**$(v, bs, \sigma; \texttt{EDB})$: The protocol runs between a client and a server. The client inputs a value v ($v \in R$), a state σ and a bit string bs[3]. The client updates each keyword $w \in \texttt{PT}$. For each keyword w, he/she encrypts the bit string bs by using the simple symmetric encryption with homomorphic addition to get the encrypted bit string e. To save the client storage, the one time key sk_c is generated by a hash function $H_3(K'_w, c)$, where c is the counter. Then he/she chooses a random search token and use a hash function to get the update token. He/She also uses another hash function to mask the previous search token. After that, the client sends the update token, e and the masked previous search token C to the server and update **CT** to get a new state σ'. Finally, the server outputs an updated encrypted database $\texttt{EDB}'$.

- $bs \leftarrow$ **Search**$(q, \sigma; \texttt{EDB})$: The protocol runs between a client and a server. The client inputs a range query q and a state σ, and the server inputs $\texttt{EDB}$. Firstly, the client gets $\texttt{BRC}$. For each keyword $w \in \texttt{BRC}$, he/she gets the search token corresponding to the keyword w from **CT** and generates the K_w. Then he/she sends them to the server. The server retrieves all the encrypted bit strings e corresponding to w. To reduce the communication overhead, the server adds them together by using the homomorphic addition ($\texttt{Add}$) of the simple symmetric encryption to get the final result Sum_e and sends it to the client. Finally, the client decrypts it and outputs the final bit string bs, which can be used to retrieve the matching files. Note that, in order to save the server storage, for every search, the server can remove all entries corresponding to w and store the final result Sum_e corresponding to the current search token ST_c to the $\texttt{EDB}$. Moreover, the client does not need to re-encrypt the final result bs, which makes our scheme more efficient than the one in [62].

5.6 Security Analysis

In this section, we give the security proof of our proposed scheme.

Theorem 4. *(Adaptive forward and Type-R backward privacy of $\texttt{FBDSSE-RQ}$). Let F be a secure PRF, $\Pi = (\texttt{Setup}, \texttt{Enc}, \texttt{Dec}, \texttt{Add})$ be a perfectly secure simple symmetric encryption with homomorphic addition, and H_1, H_2 and H_3 be random oracles. We define $\mathcal{L}_{\texttt{FBDSSE-RQ}} = (\mathcal{L}^{Search}_{\texttt{FBDSSE-RQ}}, \mathcal{L}^{Update}_{\texttt{FBDSSE-RQ}})$,*

[3]Note that, we can update many file identifiers through one update query by using bit string representation bs.

where $\mathcal{L}^{Search}_{FBDSSE-RQ}(q) = (\mathtt{sp}(q), \mathtt{rp}(q), \mathtt{Time}(q))$ *and* $\mathcal{L}^{Update}_{FBDSSE-RQ}(op, v, bs) = \mathcal{L}(\ell{+}1)$. *Then* FBDSSE−RQ *is* $\mathcal{L}_{FBDSSE-RQ}$*-adaptively forward and Type-R backward private.*

Proof. Similar to the proof from [63], we formulate a sequence of games from DSSEREAL to DSSEIDEAL. We show that every two consecutive games are indistinguishable. Finally, we simulate DSSEIDEAL with the leakage functions defined in **Theorem 4**.

Game G_0: G_0 is exactly same as the real world game $\mathtt{DSSEREAL}^{\mathtt{FBDSSE-RQ}}_{\mathcal{A}}(\lambda)$. So we can write that

$$\Pr[\mathtt{DSSEREAL}^{\mathtt{FBDSSE-RQ}}_{\mathcal{A}}(\lambda) = 1] = \Pr[G_0 = 1].$$

Game G_1: Instead of the generation of a key for keyword w using F, we choose the key at random and with uniform probability. The key and the corresponding keyword are stored in the table Key. If a keyword has been queried, then the corresponding key is fetched from the table Key. Assuming that an adversary $\mathcal{A}$ is able to distinguish between G_0 and G_1, then we can build an adversary $\mathcal{B}_1$ to distinguish between F and a truly random function. More formally,

$$\Pr[G_0 = 1] - \Pr[G_1 = 1] \leq \mathbf{Adv}^{\mathtt{prf}}_{F,\mathcal{B}_1}(\lambda).$$

Game G_2: The game is described in Algorithm 13. For the **Update** protocol, an update token UT is picked randomly and is stored in the table UT. When the **Search** protocol is called, the random tokens are generated by the random oracle H_1 such that $H_1(K_w, ST_c) = \mathtt{UT}[w, c]$. The value (K_w, ST_c) is stored in table $\mathtt{H}_1$ for future queries. If an entry (K_w, ST_{c+1}) already in table $\mathtt{H}_1$, then we cannot obtain the requested equality $H_1(K_w, ST_{c+1}) = \mathtt{UT}[w, c{+}1]$ and the game aborts. Now, we show that the abortion possibility is negligible. As a search token is chosen randomly, the probability of a correct guess for search token ST_{c+1} by the adversary is $1/2^\lambda$. If $\mathcal{A}$ makes polynomial number $p(\lambda)$ of queries, then

$$\Pr[G_1 = 1] - \Pr[G_2 = 1] \leq p(\lambda)/2^\lambda$$

Game G_3: We model the H_2 as a random oracle which is similar to H_1 in G_2. So we can write

$$\Pr[G_2 = 1] - \Pr[G_3 = 1] \leq p(\lambda)/2^\lambda$$

Game G_4: Again, we model the H_3 as a random oracle. If the adversary does not know the key K'_w, then the probability of guessing the right key is $1/2^\lambda$ (we set the length of K'_w to λ). Assuming that $\mathcal{A}$ makes polynomial number $p(\lambda)$ of queries, the probability is $p(\lambda)/2^\lambda$. So we have

Algorithm 13 G_2 for FBDSSE-RQ

Setup(1^λ)

Client:

$K \xleftarrow{\$} \{0,1\}^\lambda$, $n \leftarrow \texttt{Setup}(1^\lambda)$
CT, $\texttt{EDB} \leftarrow$ empty map
Set the range boundary d.
return $(\texttt{EDB}, \sigma = (n, d, K, \textbf{CT}))$

Update$(v, bs, \sigma; \texttt{EDB})$ $\triangleright\ 0 \le v < d$

Client:

$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{PT} \leftarrow \texttt{TPath}(v, \texttt{BT})$
for $w \in \texttt{PT}$ **do**
  $K_w || K'_w \leftarrow \texttt{Key}(w)$
  $(ST_0, \cdots, ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    $c \leftarrow -1, ST_c \leftarrow \{0,1\}^\lambda$
  end if
  $ST_{c+1} \leftarrow \{0,1\}^\lambda$
  $\textbf{CT}[w] \leftarrow (ST_0, \cdots, ST_{c+1}, c+1)$
  $UT_{c+1} \leftarrow \{0,1\}^\lambda$
  $\texttt{UT}[w, c+1] \leftarrow UT_{c+1}$
  $C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
  $sk_{c+1} \leftarrow H_3(K'_w, c+1)$
  $e_{c+1} \leftarrow \texttt{Enc}(sk_{c+1}, bs, n)$
  Send $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to the server.
end for

Server:

Upon receiving $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$
Set $\texttt{EDB}[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$

Search$(q, \sigma, \texttt{EDB})$ $\triangleright\ q = [a, b]$, where $0 \le a < b \le d-1$.

Client:

$\texttt{BT} \leftarrow \texttt{TGen}(d)$
$\texttt{BRC} \leftarrow \texttt{TGetCover}(q, \texttt{BT})$
for $w \in \texttt{BRC}$ **do**
  $K_w || K'_w \leftarrow \texttt{Key}(w)$
  $(ST_0, \cdots, ST_c, c) \leftarrow \textbf{CT}[w]$
  if $(ST_c, c) = \perp$ **then**
    return $\perp$
  end if
  for $i = 0$ to c **do**
    $H_1(K_w, ST_i) \leftarrow \texttt{UT}[w, i]$
  end for
end for
Send $\{(K_w, ST_c, c)\}_{w \in \texttt{BRC}}$ to the server.

Server:

$Sum \leftarrow 0$
for each (K_w, ST_c, c) **do**
  $Sum_e \leftarrow 0$
  for $i = c$ to 0 **do**
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $(e_i, C_{ST_{i-1}}) \leftarrow \texttt{EDB}[UT_i]$
    $Sum_e \leftarrow \texttt{Add}(Sum_e, e_i, n)$
    Remove $\texttt{EDB}[UT_i]$
    if $C_{ST_{i-1}} = \perp$ **then**
      $Break$
    end if
    $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
  end for
  $\texttt{EDB}[UT_c] \leftarrow (Sum_e, \perp)$
  $Sum \leftarrow \texttt{Add}(Sum, Sum_e, n)$
end for
Send Sum to the client.

Client:

$Sum_{sk} \leftarrow 0$
for $w \in \texttt{BRC}$ **do**
  for $i = c$ to 0 **do**
    $sk_i \leftarrow H_3(K'_w, i)$
    $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
  end for
end for
$bs \leftarrow \texttt{Dec}(Sum_{sk}, Sum, n)$
return bs

$$\Pr[G_3 = 1] - \Pr[G_4 = 1] \le p(\lambda)/2^\lambda$$

Game G_5: We replace the bit string bs by the string of all zeros (its length is y). If the adversary $\mathcal{A}$ is able to distinguish between G_5 and G_4, then we can build a reduction $\mathcal{B}_2$ to break the perfect security of the simple symmetric encryption with homomorphic addition II. So we have

Algorithm 14 Simulator $\mathcal{S}$ for $\texttt{FBDSSE-RQ}$

$\mathcal{S}$.**Setup**(1^λ)

$n \leftarrow \texttt{Setup}(1^\lambda)$
Set the range boundary d.
CT, $\texttt{EDB} \leftarrow$ empty map
return $(\texttt{EDB}, \mathbf{CT}, n, d)$

$\mathcal{S}$.**Update**$(\ell + 1)$

Client:

for 0 to ℓ **do**
  $\texttt{UT}[t] \leftarrow \{0,1\}^\lambda$
  $\texttt{C}[t] \leftarrow \{0,1\}^\lambda$
  $\texttt{sk}[t] \leftarrow \{0,1\}^\lambda$
  $\texttt{e}[t] \leftarrow \texttt{Enc}(\texttt{sk}[t], 0s, n)$
  Send $(\texttt{UT}[t], (\texttt{e}[t], \texttt{C}[t]))$ to the server.
  $t \leftarrow t + 1$
end for

$\mathcal{S}$.**Search**$(\texttt{sp}(q), \texttt{rp}(q), \texttt{Time}(q))$

Client:

$\hat{q} \leftarrow \texttt{min sp}(q)$
$\hat{\texttt{BRC}} \leftarrow \hat{q}$
for $w \in \hat{\texttt{BRC}}$ **do**
  $K_w || K'_w \leftarrow \texttt{Key}(w)$
  $(ST_c, c) \leftarrow \mathbf{CT}[w]$
  Parse $\texttt{rp}(\hat{q})$ as $\overline{bs}$.
  Parse $\texttt{Time}(w)$ as $(t_0, \cdots, t_c)$, where $\texttt{Time}(w) \in \texttt{Time}(\hat{q})$.
  if $(ST_c, c) = \perp$ **then**
    return $\perp$
  end if
  for $i = c$ to 0 **do**
    $ST_{i-1} \leftarrow \{0,1\}^\lambda$
    Program $H_1(K_w, ST_i) \leftarrow \texttt{UT}[t_i]$
    Program $H_2(K_w, ST_i) \leftarrow \texttt{C}[t_i] \oplus ST_{i-1}$
    if $i = c$ and w is the last keyword in $\hat{\texttt{BRC}}$ **then**
      Program $H_3(K'_w, i) \leftarrow \texttt{sk}[t_i] - \overline{bs}$
    else
      Program $H_3(K'_w, i) \leftarrow \texttt{sk}[t_i]$
    end if
  end for
end for
Send $\{(K_w, ST_c, c)\}_{w \in \hat{\texttt{BRC}}}$ to the server.

$$\Pr[G_4 = 1] - \Pr[G_5 = 1] \leq \mathbf{Adv}^{\texttt{PS}}_{\Pi, \mathcal{B}_2}(\lambda).$$

Simulator Now we can replace the searched range query q with $\texttt{sp}(q)$ in G_5 to simulate the ideal world in Algorithm 14, it uses the first timestamp $\hat{q} \leftarrow \texttt{min sp}(q)$ for the range query q. We ignore a part of Algorithm 13 which does not influence the view of the adversary.

We are ready to show that G_5 and **Simulator** are indistinguishable. For **Update**, it is obvious since we choose new random strings for each update in G_5. For **Search**, the simulator starts from the current search token ST_c and choose a random string for previous search token. Then it embeds it to the ciphertext C through H_2. Moreover, $\mathcal{S}$ embeds the $\overline{bs}$ to the ST_c of the last keyword in $\texttt{BRC}$ and all 0s to the remaining search tokens through H_3. Finally, we map the pairs (w, i) to the global update count t. Then we can map the values in the table $\texttt{UT}$, $\texttt{C}$ and $\texttt{sk}$ that we chose randomly in **Update** to the corresponding values for the pair (w, i) in the **Search**. Hence,

$$\Pr[G_5 = 1] = \Pr[\texttt{DSSEIDEAL}^{\texttt{FBDSSE-RQ}}_{\mathcal{A}, \mathcal{S}}(\lambda) = 1]$$

Finally,

$$\Pr[\mathtt{DSSEREAL}_{\mathcal{A}}^{\mathtt{FBDSSE-RQ}}(\lambda) = 1] - \Pr[\mathtt{DSSEIDEAL}_{\mathcal{A},\mathcal{S}}^{\mathtt{FBDSSE-RQ}}(\lambda) = 1] \leq \mathbf{Adv}_{F,\mathcal{B}_1}^{\mathtt{prf}}(\lambda) + \mathbf{Adv}_{\Pi,\mathcal{B}_2}^{\mathtt{PS}}(\lambda) + 3p(\lambda)/2^{\lambda}$$

which completes the proof.

□

5.7 Experimental Analysis

In this section, we evaluate the performance of our schemes using a testbed of one workstation. This machine plays the roles of the client and server. The hardware and software of this machine are as follows: Mac Book Pro, Intel Core i7 CPU @ 2.8GHz RAM 16GB, Java Programming Language, and macOS 10.13.2. Note that we use the bitmap index to denote file identifiers. We use the "BigInteger" with different bit lengths to denote the bitmap index with different sizes, which act as the database with different numbers of files. The relation between the i-th bit and the actual file is out of our scope. The update time includes the client token generation time and server update time. The search time includes the token generation time, the server search time, and the client decryption time. Note that the result depends on the maximum number of files supported by the system (the bit length) only.

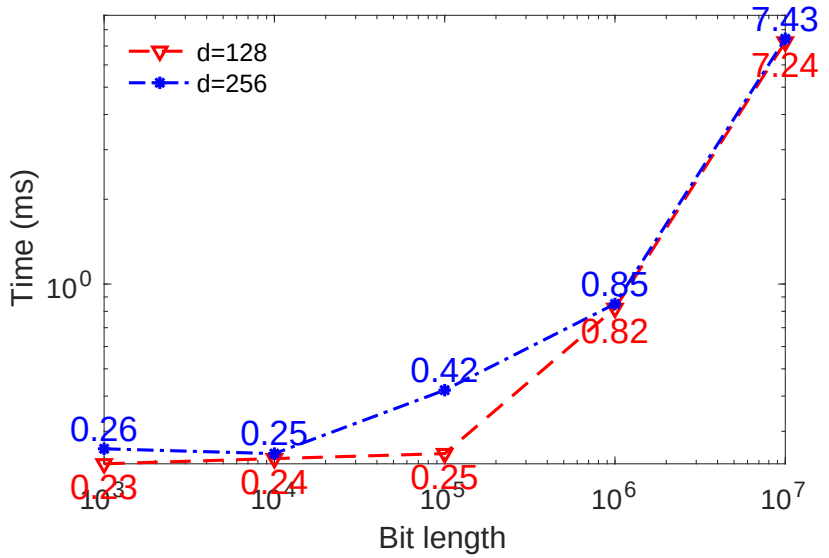


Figure 5.3: The update time of FBDSSE−RQ for different bit lengths and the parameter d

Fig. 5.3 shows the update time of our scheme for different bit lengths and the parameter d. The bit length refers to y, which is equal to the maximum number of files supported by the system. The parameter d refers to the boundary of our range query. We update one time for each value. We get the total update time for all values and divide it by the number of values, so we get the average update time for each value. As the bit length increases, the update time grows (see Fig. 5.3). There is an exception when the bit length jumps from 10^3 to

10^4 (see the line for $d = 256$). This is due to the fact that modulo addition does not contribute too much when the bit string is smaller than 10^4. We also observe that the average update time for $d = 256$ is larger than the time for $d = 128$. This is because, when $d = 256$, the binary tree has more levels, which means it needs more updates than the one for $d = 128$.

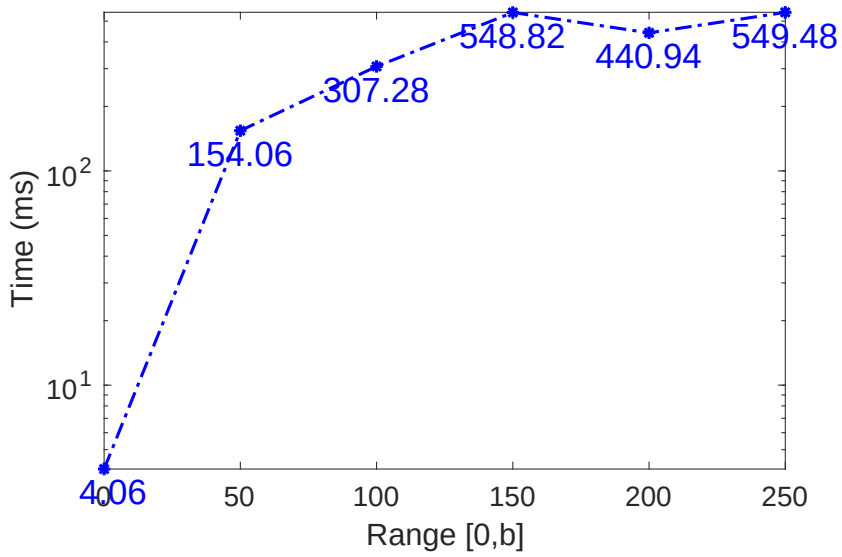


Figure 5.4: The search time of FBDSSE−RQ for different ranges ($d = 256$, bit length is 10^7)

We evaluate the search time of our scheme for different ranges ($[0, 0], [0, 50], [0, 100], [0, 150], [0, 200], [0, 250]$), where $d = 256$ and bit length is 10^7. The results are given in Fig. 5.4. It can be seen that, in general, a larger range requires a larger search time. However, this is not always true. The search time depends on the number of keywords in BRC of a range. The search time for the range $[0, 150]$ is larger than the search time for the range $[0, 200]$ because of the number of keywords in BRC for the range $[0, 150]$. In addition, with the increase of the bit length, the search time increases.

Theoretically, the bit string can be of an arbitrary length, but a larger n (e.g., $\ell = 2^{23}$) significantly increases the time needed for modulo additions. To mitigate this problem, we can divide a large bit string into several shorter ones as in the multi-block setting [63]. We refer readers to [63] for details.

5.8 Conclusion

In this chapter, we propose a forward and backward private DSSE for range queries (named FBDSSE−RQ), which requires only one roundtrip. In other words, for every search, it does not require re-encryption of the matching files, which makes our scheme more efficient. Moreover, we refine the construction of the binary tree from [54]. Names of nodes are derived from their leaf nodes, rather than from the order of node insertion [54]. In addition, we define a new backward privacy notion for our range queries called Type-R. For our range query, to update a file with value v, it leaks the number of keywords that have been updated

due to the binary tree data structure. From the security and experimental analyses, we can see that our proposed scheme achieves claimed security goals and is efficient.

Chapter 6

Future Directions

In this chapter, we are going to give the possible research directions that related to dynamic searchable symmetric encryption (DSSE).

- **Mulit-client DSSE.** Most existing (D)SSE schemes are based on the two party model (the data owner and the sever), which may not suitable for the scenario where there are many users. In 2006, Curtmola et al. [11] introduced the multi-client SSE. Later, many schemes in this area have been proposed [34, 35, 37]. In the future, we will try to extend our schemes to the multi-client setting.
- **Verifiable DSSE.** In this thesis, we assume the server is semi-honest, where the server will honestly execute all the procedures while it may be curious about the underlying encrypted data. However, in the real world, the server may try to save computation and bandwidth to return the incomplete results. To tackle this, verifiable searchable symmetric encryption schemes have been proposed [32, 33]. Later, Wang et al. [68] proposed a verifiable DSSE scheme with forward privacy. By investigating their scheme, we will try to make our schemes supporting verifiability.
- **Forward and backward private DSSE with small client storage.** To achieve forward privacy, most DSSE schemes deployed the framework of Bost et al. [20], which relies on the one-wayness of the search trapdoors. However, at the client side, these schemes need to store a counter for each keyword in the database. This increases client storage (especially for the database with large numbers of keywords). Then how to save the client storage without losing the forward and backward privacy may be another interesting research direction.
- **Secure DSSE supporting range queries.** As mentioned in Chapter 2, many researchers [29–31] leverage the access pattern [1] to recover the values of range queries. To circumvent this problem, in the future, we will try to hide the access pattern of our DSSE schemes supporting range queries.

[1] Note that, this is the "standard" leakage that first formally introduced by Curtmola et al. [11].

Chapter 7

Conclusion

This thesis shows how to build forward/backward private DSSE schemes supporting range queries and a DSSE with forward and stronger backward privacy. To achieve stronger backward privacy, we introduce the bitmap index and the simple symmetric encryption with homomorphic addition. The stronger backward privacy we achieve, named Type-I^{-}, is somewhat stronger than Type-I. To achieve range queries, we refine the construction of the binary tree from [54]. For our binary tree, we label all nodes by keywords. Names of nodes are derived from their leaf nodes rather than from the order of node insertion. We also modify algorithms for the binary tree. Moreover, we define a new backward privacy for our range queries named Type-R. Compared with single keyword queries, range queries introduce more leakages. We map a range query into several keywords that are assigned to nodes of our binary tree. For a range search query $[a, b]$, a query leaks the number of keywords, the total number of updates and update time for each keyword, the repetition of these keywords and the final results for the range query, and for the update with value v, it leaks the number of keywords that have been updated (the number of levels of the binary tree). Moreover, we introduce DSSE schemes with forward/backward privacy. Then we propose DSSE schemes with forward and Type-I^{-} backward privacy. After that, we present a forward and Type-R backward private DSSE for range queries. Finally, the security and experimentation evaluations demonstrate our schemes are secure and efficient.

References

[1] Google drive. https://www.google.com/drive/.

[2] Dropbox. www.dropbox.com.

[3] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou. Toward secure and dependable storage services in cloud computing. *Services Computing, IEEE Transactions on*, 5(2):220–232, 2012.

[4] Cong Wang, Sherman SM Chow, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for secure cloud storage. *Computers, IEEE Transactions on*, 62(2):362–375, 2013.

[5] Facebook–cambridge analytica data scandal. https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal.

[6] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[7] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 157–167. Society for Industrial and Applied Mathematics, 2012.

[8] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *International Colloquium on Automata, Languages, and Programming*, pages 803–815. Springer, 2005.

[9] Sergey Yekhanin. Private information retrieval. *Communications of the ACM*, 53(4):68–73, 2010.

[10] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[11] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *CCS06*, pages 79–88, 2006.

[12] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*, pages 353–373. Springer, 2013.

[13] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security*, pages 123–145. Springer, 2015.

[14] Cong Zuo, James Macindoe, Siyin Yang, Ron Steinfeld, and Joseph K. Liu. Trusted boolean search on cloud using searchable symmetric encryption. In *Trustcom, 2016 IEEE*, pages 113–120. IEEE, 2016.

[15] 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*, volume 14, pages 23–26. Citeseer, 2014.

[16] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 668–679. ACM, 2015.

[17] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communi-*

cations security, pages 965–976. ACM, 2012.

[18] 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 Symposium*, pages 707–720, 2016.

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

[20] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1143–1154. ACM, 2016.

[21] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1465–1482. ACM, 2017.

[22] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

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

[24] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442–455. Springer, 2005.

[25] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, and Cong Zuo. Result pattern hiding searchable encryption for conjunctive queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 745–762, 2018.

[26] Yu Zhang, Yin Li, and Yifan Wang. Conjunctive and disjunctive keyword search over encrypted mobile cloud data in public key system. *Mobile Information Systems*, 2018, 2018.

[27] Zhiqiang Wu and Kenli Li. Vbtree: forward secure conjunctive queries over encrypted data for cloud computing. *The VLDB Journal*, 28(1):25–46, 2019.

[28] Chengyu Hu, Xiangfu Song, Pengtao Liu, Yue Xin, Yuqin Xu, Yuyu Duan, and Rong Hao. Forward secure conjunctive-keyword searchable encryption. *IEEE Access*, 7:35035–35048, 2019.

[29] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[30] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314. IEEE, 2018.

[31] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1067–1083. IEEE, 2019.

[32] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *Communications (ICC), 2012 IEEE International Conference on*, pages 917–922. IEEE, 2012.

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

[34] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 577–594. Springer, 2010.

[35] Emiliano De Cristofaro, Yanbin Lu, and Gene Tsudik. Efficient techniques for privacy-preserving sharing of sensitive information. In *International Conference on Trust and Trustworthy Computing*, pages 239–253. Springer, 2011.

[36] Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Outsourced symmetric private information retrieval. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 875–888. ACM, 2013.

[37] Shi-Feng Sun, Joseph K. Liu, Amin Sakzad, Ron Steinfeld, and Tsz Hon Yuen. An efficient non-interactive multi-client searchable encryption with support for boolean queries. In *European symposium on research in computer security*, pages 154–172. Springer, 2016.

[38] Ronald Cramer and Victor Shoup. Signature schemes based on the strong rsa assumption. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):161–185, 2000.

[39] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 321–334. IEEE, 2007.

[40] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. Acm, 2006.

[41] Javier Herranz, Fabien Laguillaumie, and Carla Ràfols. Constant size ciphertexts in threshold attribute-based encryption. In *Public Key Cryptography–PKC 2010*, pages 19–34. Springer, 2010.

[42] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data*, pages 185–198. ACM, 2016.

[43] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In *European Symposium on Research in Computer Security*, pages 228–246. Springer, 2018.

[44] Shabnam Kasra Kermanshahi, Joseph K. Liu, and Ron Steinfeld. Multi-user cloud-based secure keyword search. In *Australasian Conference on Information Security and Privacy*, pages 227–247. Springer, 2017.

[45] Yunling Wang, Jianfeng Wang, Shifeng Sun, Joseph K. Liu, Willy Susilo, and Xiaofeng Chen. Towards multi-user searchable encryption supporting boolean query and fast decryption. In *ProvSec 2017*, volume 10592 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2017.

[46] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1449–1463. ACM, 2017.

[47] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2004.

[48] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. Order-preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 224–241. Springer, 2009.

[49] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011.

[50] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional en-

cryption without obfuscation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 563–594. Springer, 2015.
[51] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. Practical order-revealing encryption with limited leakage. In *International Conference on Fast Software Encryption*, pages 474–493. Springer, 2016.
[52] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS 2014*, pages 215–226. ACM, 2014.
[53] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual Cryptology Conference*, pages 563–592. Springer, 2016.
[54] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption schemes supporting range queries with forward/backward privacy. *CoRR*, abs/1905.08561, 2019.
[55] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS 2018*, pages 1038–1055. ACM, 2018.
[56] Shi-Feng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *CCS 2018*, pages 763–780. ACM, 2018.
[57] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with sgx. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.
[58] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, page 4. ACM, 2019.
[59] Vivek Sharma. Bitmap index vs. b-tree index: Which and when? *Oracle Technical Network*, 2005. http://www.oracle.com/technetwork/articles/sharma-indexes-093638.html.
[60] Claude Castelluccia, Einar Mykletun, and Gene Tsudik. Efficient aggregation of encrypted data in wireless sensor networks. 3rd intl. In *Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Sensor Networks, Italy*, 2005.
[61] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.
[62] Jiafan Wang and Sherman S. M. Chow. Forward and backward-secure range-searchable symmetric encryption. *IACR Cryptology ePrint Archive*, 2019:497, 2019.
[63] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption with forward and stronger backward privacy. In *European Symposium on Research in Computer Security*, pages 283–303. Springer, 2019.
[64] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 351–368. Springer, 2014.
[65] Ian Miers and Payman Mohassel. Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. In *NDSS*, 2017.
[66] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *ACM Transactions on Database Systems (TODS)*, 43(1):2, 2018.
[67] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, page 4. ACM, 2019.

[68] Zhongjun Zhang, Jianfeng Wang, Yunling Wang, Yaping Su, and Xiaofeng Chen. Towards efficient verifiable forward secure searchable symmetric encryption. In *Computer Security – ESORICS 2019*, pages 304–321. Springer International Publishing, 2019.