Making Cloud Storages Secure and Efficient

Submitted to Indian Statistical Institute
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in Computer Science

Avishek Majumder
Applied Statistics Unit (ASU)

Indian Statistical Institute
Kolkata - 700108, India

Dedicated to

My Family

LIST OF PUBLICATIONS/MANUSCRIPTS

1. Debrup Chakraborty, **Avishek Majumder**, and Subhabrata Samajder "Making Searchable Symmetric Encryption Schemes Smaller and Faster.", Published in International Journal of Information Security (doi.org/10.1007/s10207-024-00915-y).

2. Mohamed Ahmed Abdelraheem, Sanjay Bhattacherjee, Theo Henault, and **Avishek Majumder**. "Conjunctive Dynamic Searchable Symmetric Encryption – A Generic Framework.", The IACR Communications in Cryptology (CiC) (submitted).

ACKNOWLEDGEMENT

I would like to take this opportunity to express my sincere gratitude to all those who have contributed to the completion of my PhD thesis.

First and foremost, I am deeply grateful to my supervisor, Dr. Debrup Chakraborty, for his unwavering support, guidance, and mentorship throughout my research journey and beyond. His valuable comments, motivating discussions, and ideas greatly helped my research work. His wisdom, values and perspective towards life have helped me become a better human being.

I am indebted to Dr. Sanjay Bhattacherjee for his guidance and support. A significant part of the work reported in this thesis has been done in collaboration with him. Furthermore, Dr. Bhattacherjee had a significant advisory role throughout my tenure as a graduate student in ISI.

A part of the thesis was done in collaboration with Dr. Subhabrata Samajder. Who was always available to listen to my ideas and enrich me with his valuable comments. I'm thankful to him for his mentorship, and I am also indebted to him for the support he provided during my visit to IIIT Delhi.

I am sincerely grateful to Prof. Bimal Kumar Roy, Prof. Palash Sarkar, Prof. Mridul Nandi, Prof. Subhamoy Maitra, Prof. Sanjit Chatterjee, and Dr. Sabyasachi Karati for their invaluable help and guidance.

I am thankful to Dr. Sayantan Mukherjee, Dr. Nilanjan Datta, Dr. Avijit Dutta, Dr. Mohamed Ahmed Abdelraheem, and Dr. Dhiman Saha, who have helped me with their noble guidance, support, and encouragement as a senior and collaborator.

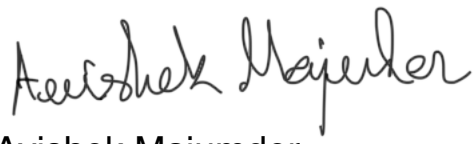
I would also like to thank Dr. Laltu Sardar, Mr. Bibhash Chanda Das, and Mr. Theo Henault for a wonderful journey of collaboration. I would also like to thank Sourav for helping me with my work. I would like to express my love and respect to Dr. Sayan Dey for his believe in me and continuous support and motivation throughout my journey, starting from my M.Tech.

I thank Samir for being with me in my PhD journey. I could not thank you enough to Sougata for always being there for me and encouraging me in my low times. My

journey toward a PhD would not be possible without the friendship and support of Rakesh and Chotu. Also, my sincere love and respect to Diptendu-da for always listening to me in my low time and always encouraging me.

I am also thankful to the former and current members of our research group, including Mostaf, Prabal, Nayana, Manab, Anurag, Subhra, Subhadip-da, Aniruddha, Laltu-da, Amit, Soumya, Pritam, Biswajit, Suprita, Ananda, Susanta, Aniruddha, Jyotirmoy, Animesh, Chandranan, Gourab, Anup, Suman, Suvo, Manish, Pravat with whom I have always had fruitful discussions and enjoyable times.

I would like to express my heartfelt gratitude to my entire family for their unwavering support and guidance. My dream of pursuing research would be incomplete without my parents' constant support and encouragement, Mr. Bibek Jyoti Majumdar, Mrs. Tanusree Majumdar, and sister Arijita Majumder. Last but not least, I am grateful to my wife, Koushani, for her support, which has been a constant source of inspiration for me for the past two years.



Avishek Majumder

ABSTRACT

Over the years, *searchable symmetric encryption* (SSE) schemes have emerged as a promising tool for enabling efficient query processing over encrypted data stored in untrusted cloud servers. This thesis mainly focuses on efficiency and security enhancements of *dynamic searchable symmetric encryption* (DSSE) schemes, which support various query types and are secure against several adversarial conditions.

For any SSE scheme, its query processing, storage, and communication costs are directly related to the size of the encrypted index stored on the server. A reduction of the index size naturally leads to enhanced search efficiency and reduced storage and communication costs. We are unaware of any previous attempts to reduce the index size of SSE schemes. We introduce a novel technique to directly reduce the index size of any SSE. Our proposed method generically transforms any secure single keyword SSE into an equivalently functional and secure version with reduced storage requirements, resulting in faster search and reduced communication overhead. Our technique involves arranging the set of document identifiers $\mathsf{db}(w)$ related to a keyword w in the leaf nodes of a complete binary tree, eventually obtaining a succinct representation of the set $\mathsf{db}(w)$. This compact representation leads to smaller index sizes. We conduct extensive theoretical analysis to prove the correctness of our scheme. Additionally, our experiments on real and synthetic data validate the effectiveness of our approach and demonstrate its practical applicability.

Among the few SSE schemes available in the literature which support complex query types like conjunctive queries, the *oblivious cross tag* (OXT) scheme from Crypto'13 is the most efficient one. OXT has the limitation that it only works for static databases. In NDSS'20, an extension of OXT called the *oblivious dynamic cross tag* (ODXT) was proposed. ODTX supports conjunctive queries with dynamic updates. However, ODXT is not *forward private*.

We propose a generic framework for designing *conjunctive dynamic SSE* (CDSSE) schemes, supporting conjunctive queries that allow dynamic updates while being both forward and backward private simultaneously. To the best of our knowledge such a

scheme does not exist till date. Our scheme assumes a restricted update model where a document with its associated keywords can be dynamically added to or deleted from the database as a whole, but the set of keywords for a document is not modified once uploaded. We define forward and backward privacy for this new setting of updates and extend the OXT scheme to make it dynamic in the new setting. We prove the security of our construction against adaptive adversaries and analyse the precise leakages to the adversarial server. Experiments show that our schemes are very efficient.

Another less studied aspect of SSE schemes is verifiability. In an SSE scheme, the server may be dishonest and may not respond to a client's queries following the prescribed protocol. A verifiable SSE can detect such anomalous behaviour of a server. To defend against such malicious adversaries, previous approaches employ *authenticated encryption* (AE) to furnish a "*proof*" for each update. We propose a new construction where we convert any forward and backward private adaptively secure SSE scheme into a verifiable SSE. Our construction uses a new class of message authentication codes (MAC), which we call *updatable message authentication codes* (UdMAC). A UdMAC allows the verification tag for a message to be updated with each modification to the message without recomputing the entire MAC, ensuring efficiency. We establish security requirements for such a MAC and introduce two constructions, ConCatU and XoRU, which work with two different types of message updates, namely, concatenation and exclusive-or (XOR), respectively. Furthermore, we present the first generic construction for a forward and backward private *fault-tolerant verifiable* DSSE using a UdMAC construction and prove its security. Our construction converts any generic forward and backward secure SSE secure in an honest-but-curious adversarial model into an equivalently secure DSSE secure in a malicious adversarial model with faulty updates.

Contents

1 Introduction **17**

1.1 Searchable Symmetric Encryption 18

1.2 Scope of This Thesis 21

1.3 The Road-Map 22

2 Preliminaries **27**

2.1 General Notations 27

2.2 Hardness Assumptions 28

2.3 Dynamic Searchable Symmetric Encryption 30

2.4 Conjunctive Dynamic SSE 33

2.5 Security of DSSE 33

2.5.1 Security Game 35

2.5.2 Correctness of DSSE 36

2.5.3 Soundness of DSSE 37

2.5.4 Fault-tolerant Verifiable DSSE 39

2.5.5 Forward and Backward Privacy (Single Keyword DSSE) . . . 39

2.6 Related Works 43

2.7 Final Remarks 48

3 Tree Covers **49**

3.1 Binary Trees and Tree Covers . . . 51
3.1.1 Binary Trees . . . 51
3.1.2 Tree Cover . . . 52
3.1.3 Cover of a Configuration . . . 54
3.1.4 Cover of a Dynamic Configuration . . . 56
3.2 Cover Generation Algorithms . . . 58
3.2.1 Pure Cover: A Tree Cover Scheme for Pure Configurations . . . 58
3.2.2 Mixed Cover: Generating a Smaller Sized Cover . . . 61
3.2.3 Complexity of Proposed Algorithms . . . 71
3.2.4 Dynamic Tree Cover Scheme . . . 72
3.3 Expected Cover Size of a Pure Cover . . . 74
3.4 Experimental Results . . . 77
3.5 Final Remarks . . . 79

4 Tree Cover Based SSE **81**
4.1 Constructing Static SSE Using Tree Cover . . . 82
4.1.1 Cover-based Representation of DB. . . . 83
4.1.2 Generic Static SSE Using Keyword Cover . . . 84
4.2 Constructing Dynamic SSE Using Tree Cover . . . 87
4.3 Discussions . . . 92
4.4 Security of $d\Sigma$ . . . 94
4.5 Experimental Results . . . 99
4.5.1 Extra Overheads . . . 99
4.6 Final Remarks . . . 102

5 Conjunctive Dynamic SSE – A Generic Framework **105**
5.1 Non-modifiable Database . . . 106
5.2 Forward and Backward Privacy in CDSSE for Non-modifiable Documents 108
5.3 Our Construction . . . 111
5.3.1 Our Generic CDSSE Scheme . . . 112
5.3.2 Design Rational . . . 113

5.4 Security of Our Construction . . . 116
5.5 Leakage Analysis . . . 122
5.6 Implementation and Experimental Results . . . 130
5.7 Final Remarks . . . 133

6 Updatable Message Authentication Codes 135
6.1 Message Authentication Codes . . . 136
6.2 Polynomial MACs . . . 137
6.3 Updatable MACs . . . 139
6.3.1 Security of Updatable MACs . . . 141
6.3.2 Some New Notations . . . 142
6.4 ConCatU: An Updatable MAC for Concatenation . . . 142
6.4.1 Instantiations and Efficiency . . . 145
6.4.2 Security of ConCatU . . . 147
6.5 XoRU: An Updatable MAC for $\oplus$ Difference . . . 154
6.5.1 Efficiency of XoRU . . . 156
6.5.2 Security of XoRU . . . 158
6.6 Final Remarks . . . 161

7 Fault-tolerant Verifiable DSSE 163
7.1 Generic Fault-tolerant Verifiable DSSE . . . 164
7.1.1 A Generic FVDSSE Scheme vΣ Using Ψ and Σ . . . 165
7.1.2 Fault-tolerance and Correctness of the Search Result . . . 168
7.2 Comparisons and Actual Overheads . . . 169
7.3 Security . . . 172
7.3.1 Adaptive Security of vΣ . . . 172
7.3.2 Forward and Backward Privacy of vΣ . . . 175
7.3.3 Soundness of vΣ . . . 175
7.4 Final Remark . . . 178

8 Conclusion 179

List of Figures

2-1 Correctness Game of DSSE. 36

2-2 Soundness Game of DSSE 38

3-1 The tree $\mathcal{T}$ used in Example 1. 53

3-2 The coloring scheme for pure configurations 59

3-3 Pure cover generation 60

3-4 Pure cover reconstruction 61

3-5 Coloring scheme for mixed covers 62

3-6 An example of mixed cover: The leaf nodes represented by circles are assumed to bear the sign $+$, the leaf nodes represented by squares bear the sign $-$. The shaded circles represent nodes colored green and the shaded squares are nodes colored red. 63

3-7 Mixed cover generation algorithm 68

3-8 Mixed cover reconstruction 71

3-9 Dynamic cover generation 73

3-10 Dynamic cover reconstruction 74

3-11 Number of Identifiers vs Average Cover Size 79

3-12 Expected Cover Size Comparison 80

4-1 DB to $\widetilde{\mathsf{DB}}$ conversion algorithm 83

4-2 Generic static SSE using tree-cover scheme 85

4-3 Buffer update algorithm 86
4-4 A generic dynamic SSE using tree-cover scheme setup phase and DB to $\widetilde{\mathsf{DB}}$ conversion algorithm 87
4-5 A generic dynamic SSE using tree-cover scheme update phase 89
4-6 A generic dynamic SSE using tree-cover scheme search phase 91
4-7 A pictorial depiction of the data in Table 4.2 showing the number of tuples in the original database (represented by filled circles) and the number of tuples in the converted database (represented by filled triangles) for different database sizes. 101
4-8 Database sizes in dynamic scenario. 102

5-1 A generic dynamic single-keyword SSE (DSSE) construction Σ 112
5-2 A modular description of Γ without the SKSSE part of [29] 113
5-3 Our generic CDSSE construction Π. 114
5-4 Search time comparison between the three instantiations of our generic CDSSE for fixed $|\text{Updates}(w_2)| = 10^5$ with varying $|\text{Updates}(w_1)|$ from 10^2 to 10^5. 132
5-5 Search time comparison between the three instantiations of our generic CDSSE for fixed $|\text{Updates}(w_1)| = 10$ with varying $|\text{Updates}(w_2)|$ from 10 to 10^5. 132

6-1 The procedure $\mathsf{SeqUpdt}$ used to define correctness of an updatable MAC scheme Ψ. 140
6-2 Specification of $\mathsf{ConCatU}$ using a prf $f_K : \{0,1\}^* \to \{0,1\}^n$. 144
6-3 Specification of $\mathsf{XoRU.MacUpdate}$ and $\mathsf{XoRU.MacVerify}$. 156

7-1 Setup phase of $\mathsf{v}\Sigma$. 166
7-2 Update protocol of $\mathsf{v}\Sigma$. 166
7-3 Search protocol of $\mathsf{v}\Sigma$. 167
7-4 Verify algorithm of $\mathsf{v}\Sigma$. 168
7-5 Forger game of FVDSSE. 177

List of Tables

4.1 Characteristics of some existing SSE schemes: N is the number of (id, w) pairs, $|\mathcal{W}|$ is the number of keywords, i_w and d_w are the number of insertions and deletions, and $u_w = i_w + d_w$ is the total number of updates, and $n_w = i_w - d_w$ is the number of keyword-identifier pairs currently matching the keyword w. RT is the number of roundtrips for a search query. 95

4.2 The size of original database DB and the converted database $\widetilde{\mathsf{DB}}$ in case of Enron data. 100

4.3 Cover generation and reconstruction times. 101

5.1 Comparison of our construction with previous schemes. W: the set of keywords. D: Set of documents. W_d: Number of keywords in the document to be updated. N: Number of identifier-keyword pairs in the database. n: Number of keywords in a conjunction. u_i: Updates performed for keyword w_i. $u = \sum_{i=1}^{n} u_i$: Total updates for a conjunction of n keywords. v: Keywords involved in the edit operation. h: Average number of documents matched by any two keywords. $\mathsf{db}(\omega)$: Set of matching document identifiers for query $\omega = w_1 \wedge \cdots \wedge w_n$, where $n \geq 1$. RT: Round-Trip. FP: Forward Private. BP: Backward Private. KPRP: Keyword-Pair Result Pattern Hiding. NMD: Non-Modifiable Documents. 115

5.2 Storage cost and time required for each update. 131

5.3 Search time comparison for fixed $|\mathsf{Updates}(w_2)| = 10^5$ with varying $|\mathsf{Updates}(w_1)|$ from 10^2 to 10^5 133

7.1 Comparison of our schemes with existing VDSSE schemes. FP, BP and FT stand for forward privacy, backward privacy, and fault-tolerant, respectively. 'Computation' and 'Communication' refer to the computational and communication complexity, respectively. W is the number of distinct keywords in the database, and $|D|$ denotes the number of documents in the database. N is the number of keyword-identifier pairs. $0 < \epsilon < 1$ is a fixed constant, and α denotes the number of times the queried keyword was historically added to the database. m denotes the number of matching entries for the searched keyword. . . 170

1 Introduction

Delegating an organization's or individual's storage requirements to a third-party server has become a common practice. Storage as a Service, commonly referred to as cloud storage, is a model in which providers offer storage solutions to clients. These solutions are advantageous as they alleviate users from costs associated with storage-related hardware, software, and maintenance. While outsourcing storage requirements offers convenience and cost-effectiveness, it also raises significant security concerns. Data protection laws like the General Data Protection Regulation (GDPR) which is in force in the European Union and the European Economic Area, require that online service providers using third-party cloud services for storing data must ensure user privacy "by design and by default"[1]. Similar policies are applicable to many other countries also. This makes the problem of securing the data stored in third party servers even more important, both from the perspective of the user and the service providers.

Two main dimensions of information security are data confidentiality and integrity/authenticity. These are applicable in the context of outsourced data also. For outsourced data, confidentiality means that the data stored in the server should be unreadable to the server. Integrity means that if the server deviates from the predefined protocols for answering the client's queries and/or updating the data, then the client should be able to detect such malicious behaviour of the server. Note that, once the database is outsourced, the client can no longer monitor its current status. There-

[1] As in Art. 25 of GDPR as on January 21, 2025. https://gdpr-info.eu/art-25-gdpr/

fore, it is essential to ensure that the server accurately stores the database, executes queries, and responds properly to user queries. Thus, the problem of authentication in the case of outsourced data is as important as the problem of confidentiality. Note that in this thesis we use both the terms data integrity and authentication interchangibly.

Historically, the challenge of maintaining data confidentiality has been addressed through encryption. Secure encryption algorithms ensure that the resulting ciphertext is indistinguishable from random strings, rendering it practically unreadable for the server. However, this solution presents a dilemma, as users also require the ability to query and update the delegated data. Encrypting the stored data with a traditional encryption algorithm would hinder the server's capability to execute queries or updates on behalf of the client. A well accepted solution to this dilemma is provided by Searchable Symmetric Encryption (SSE) schemes, which are specialized encryption schemes that allow search and updates on encrypted data without compromising its confidentiality.

The traditional solution to the problem of data integrity/authentication in the symmetric key setting has been provided by Message Authentication Codes (MAC) and/or several variants of Authenticated Encryption (AE). These solutions, though important in the context of outsourced data also, have to be properly tailored to achieve integrity in data stored in the cloud.

In this thesis, we study several dimensions of the problem of securing outsourced data and provide some efficient and secure solutions. In this Chapter, we begin with a token introduction to SSE. Further, we discuss the scope of this thesis and finally provide a chapter wise summary of the rest of the thesis.

1.1 Searchable Symmetric Encryption

An SSE scheme abstracts a database as a set of documents denoted as $\mathcal{D}$. Each document $\mathsf{d}_i \in \mathcal{D}$ is associated with a unique identifier $\mathsf{id}_i \in \mathcal{I}$. These documents are viewed as collections of keywords from a predefined finite set, denoted as $\mathcal{W}$. For each

keyword, $w \in \mathcal{W}$, $\mathsf{db}(w)$ represents the set of identifiers of those documents containing the keyword w. For a $w \in \mathcal{W}$, let $t_w = \mathsf{db}(w) \times \{w\} = \{(\mathsf{id}, w) : \mathsf{id} \in \mathsf{db}(w)\}$. The database DB of an SSE scheme is viewed as the set of tuples $\mathsf{DB} = \bigcup_{w \in \mathcal{W}} t_w$.

The functionality of searching and updating encrypted data is commonly achieved through an *inverted index* [76] that stores mappings of document identifiers with keywords. Searchable Symmetric Encryption (SSE) follows a similar approach, storing the inverted index in an encrypted form so that user queries are hidden from the server, thus preserving user privacy.

Specifically, an SSE encrypts the database DB, comprising keyword-identifier pairs, using a specialized structure often referred to as an "*encrypted multi-map*" [35] (EMM) or an inverted index. This encrypted multi-map is stored on the server and facilitates both searching and updating the encrypted database. An SSE scheme typically supports three types of operations: addition, search, and deletion, which we denote by add, srch, and del, respectively. To initiate a keyword search, the client provides a *search token* for the encrypted multi-map to the server. For updates (such as additions and deletions to/from the database), the client supplies an *update token* to the server, enabling modifications to the encrypted multi-map. Subsequently, these search and update tokens are used by the server to transmit encrypted search results to the client and update the database as needed, respectively. The storage of an inverted index in an encrypted form in an untrusted server incurs extra computation and communication overhead on both the client and server. In general, this overhead is directly related to the size of the inverted index.

SSE schemes that support only the search operation are referred to as *static SSE*, whereas those SSEs that support both search and updates (addition and deletion) are known as *dynamic SSE* (DSSE) schemes. In the context of SSE, the input for an update protocol is a keyword-identifier pair (id, w) with an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$. For the search operation ($\mathsf{op} = \mathsf{srch}$), the input can either be a single keyword or multiple keywords with a specific search rule. The output of a search is a set of document identifiers that match the query rule [60, 28]. This design allows SSE to decouple the storage of documents from the storage of the data structures used for

searching and updating the database.

Most SSE schemes to date [86, 42, 60, 28, 23, 25, 87, 49, 38, 89, 90] support equality search on single keywords in each query. However, some can also support richer queries like conjunctive or boolean queries [29, 73, 47, 58, 93, 11, 67], wild card search [85, 22, 54, 37, 96, 69], and sub-string search [36, 53, 72].

Depending upon the adversarial nature of the cloud service provider platform (CSP), an SSE adversary can be classified into two categories.

a. **Honest-but-curious:** where the adversary follows the protocol honestly but attempts to gain knowledge about user data from the information that queries inadvertently reveal.

b. **Malicious:** where the adversary tries to learn about the user's data and may also cheat by either not storing the user data to save storage or not responding to a query correctly to save computation.

The primary goal of an SSE scheme is to protect users' data from both types of adversaries mentioned above, but all existing SSEs do not provide security against both these types of adversaries.

Security of SSE schemes is an evolving topic of study and is not yet fully understood. Security guarantees are generally provided in a pre-defined security model, and as it is common in the cryptographic literature, the security of an SSE scheme is reduced to the security of one or more well studied cryptographic primitives or hardness assumptions. The usefulness of such security guarantees depends heavily on how well the security model represents the real adversarial threat. There have been several instances that a provably secure SSE scheme has been attacked as the security model failed to incorporate a real attack scenario [55, 68, 27, 97].

Over the years, two key security notions in DSSE, namely, *forward privacy* and *backward privacy* have been proposed [28], which are acceptable to the community and are widely believed to encompass a large class of real attack scenarios. Intuitively, forward private SSE schemes ensure that the server cannot link new updates with previous search operations performed on the inverted index. This protects queries

made to the inverted index with newly injected files by the adversary [97]. Backward privacy ensures that after a pair (id, w) has been deleted, the identifier id will not be leaked in future searches for w. The first security definition and a scheme achieving them were proposed in [23], and the first formal definition of backward security was proposed in [25].

The major focus, till date, has been in designing SSEs secure against honest-but-curious adversaries [86, 42, 60, 28, 23, 25, 87, 49, 38, 89, 90]. A handful of studies on SSE have focused towards the adversary being malicious. A *verifiable* SSE (VSSE) scheme prevents a malicious/dishonest server from forging the client by providing a "*proof*" for every search result. Using the proof, a user can verify the correctness of the response to a search query returned by the server [63, 30, 64, 65, 24, 99, 98, 52, 48, 71, 94].

While the works mentioned above take into account a dishonest server, very few of them can handle inappropriate behaviour from the client. The client, who is unaware of the current state of the database maintained by the server, might submit inappropriate or invalid updates. For example, a client may attempt to re-insert an already existing keyword-identifier pair or may attempt to delete a nonexistent keyword-identifier pair. An SSE scheme that can handle such malicious updates from a client and still provide correct search results is referred to as a *fault-tolerant* SSE.

1.2 Scope of This Thesis

This thesis aims to *make cloud storage secure and efficient*. We specifically answer the following main questions:

a. Is it possible to develop a method for storing the encrypted inverted index in a more succinct manner, thereby reducing the computation, communication, and storage overhead of state-of-the-art SSE schemes?

b. Is it possible to efficiently support complex queries, such as conjunctive searches, in SSE while maintaining the required security and efficiency?

c. Is it possible to develop an SSE scheme that ensures correct search results while maintaining required security, even in the presence of a malicious server?

d. Is it possible to maintain the desired security and correctness of SSE even if the client performs faulty updates?

In this thesis, we answer all these questions affirmatively. We propose new constructions of SSE schemes and propose modifications over existing schemes to achieve our goals. We prove the correctness and security of our proposals and, in some cases, provide extensive experimental data to validate the utility of our proposals in real-world applications. We always consider a single user model.

1.3 The Road-Map

In this Section, we provide a short overview of our findings and a chapter-wise summary of the rest of the thesis.

Chapter 2 does not contain any new material. We begin by introducing the general notations used throughout the thesis and defining the hardness assumptions necessary to prove the security of our proposed constructions. Following this, we present the formal definition of SSE and the related security notions. Finally, we discuss some related works relevant to this thesis.

Chapter 3 and **Chapter 4** presents our work related to making SSEs efficient in terms of storage, computation and communication overheads. Our main observation is that all existing SSE schemes need to store all the keyword-identifier pairs present in the database DB. Thus, the size of the encrypted index is never smaller than the size of DB, i.e. $|\mathsf{DB}|$. In most cases, it is much bigger than $|\mathsf{DB}|$. We propose a generic pre-processing step that can be applied to any secure SSE to obtain a new SSE, which will have a significant reduction of its index size on average but would retain the security of the original scheme. This reduction in size further reduces search time and communication requirements. The main tool we use to achieve this reduction is a novel concept that we call the *tree cover*. For each keyword w we construct a complete binary tree where the number of leaf nodes is at least $|\mathcal{I}|$ and

associate each $\mathsf{id} \in \mathcal{I}$ with a leaf node. We label a node associated with id with $+$ if $\mathsf{id} \in \mathsf{db}(w)$ and with $-$ if $\mathsf{id} \notin \mathsf{db}(w)$. This labeled tree uniquely represents the set $\mathsf{db}(w)$. We show that this labeled tree can be represented by a few numbers of tree nodes, and we can use only these nodes to represent the set $\mathsf{db}(w)$ for all keywords w. This leads to a drastic reduction in the representation of $\mathsf{db}(w)$. We further use this representation of $\mathsf{db}(w)$ to design an SSE with a reduced index size.

In Chapter 3 we introduce the idea of tree covers and discuss several algorithms to generate covers. We prove the correctness of our algorithms and present theoretical estimates on cover sizes. Finally, we present experiments to validate our theoretical estimates of the cover sizes. The characterization of tree covers and the algorithms for cover generation are further used for the construction of SSEs in Chapter 4. The material presented in Chapter 3 has little reference to SSEs. It just states and solves a combinatorial problem which can be of independent interest.

In Chapter 4 we use the tools developed in Chapter 3 to construct an SSE. We treat the cases of static and dynamic SSE separately. The main result presented in this chapter is that if we take any secure DSSE Λ, then we can convert it into an equivalently functional and secure DSSE, which will have much smaller index sizes on average. We prove the security of our construction and present extensive experimental data on real data sets to demonstrate the effectiveness of our scheme.

Chapter 5 deals with SSEs, which support conjunctive search queries. The most notable existing construction of an SSE supporting conjunctive queries is the *Oblivious Cross Tag* (OXT) construction [29]. OXT supports conjunctive and general boolean queries very efficiently in large databases with reasonable leakage. However, the scheme is limited to static databases. Designing dynamic databases that support conjunctive queries while being both forward and backward private has been a long-standing open problem. The first claim of such a construction was reported in [80]. A subsequent work [100] demonstrated that the scheme in [80] fails to provide forward privacy in certain cases. Since then, several works [102, 100, 95] have proposed solutions that support conjunctive queries with forward and backward privacy, but none have achieved the efficiency close to the OXT scheme of [29].

In Chapter 5, we address the problem for a specific class of databases that we refer to as *non-modifiable databases*, which are databases where an existing document cannot be modified though new documents can be added and existing documents can be deleted. The only way to modify a document is to delete the document and then add the modified document. This model of database falls in between static databases and dynamic databases. We argue that there are numerous practical scenarios where such a model is applicable. Further, we introduce a generic technique to design an SSE scheme supporting conjunctive queries for non-modifiable databases, achieving efficiency comparable to the OXT scheme. Our approach integrates any DSSE scheme that supports equality queries and addition operations with the OXT framework of [29] in a modular, black-box fashion. We prove our scheme to be both forward and backward private if applied to non-modifiable databases.

Chapter 6 and **Chapter 7** deal with the problem of authenticity. We consider a malicious adversarial setting, where the server not only tries to learn about user data but also differs to follow the protocol correctly. Moreover, the client, being unaware of the current state of the database, may issue incorrect updates to the database. As already stated, in such scenarios, it is crucial to have cryptographic assurance that the query responses returned by the CSP are accurate.

In Chapter 6, we develop a new class of *message authentication codes* (MAC) which we call Updatable MACs (UdMAC). Let $\mathcal{M}$ be a message space. We fix $\Delta : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ to be an associative binary operator, which we call the update function. As usual, for $M, m \in \mathcal{M}$ we will denote $\Delta(M, m)$ by $M\Delta m$. For an example, one can consider Δ to be the concatenation operator, ie, $M\Delta m$ represents the message $M\|m$. A Δ-UdMAC is a stateful procedure, which has the following interesting property. Let $M \in \mathcal{M}$ be a message and let its tag be t, then one can compute the tag corresponding to the message $M\Delta m$ with only access to m, t and some state information. Thus, to outsource a message M, a Δ-UdMAC computes a tag t of M. M is sent to the server, and a tag t, along with some state information, is retained with the client. Further, suppose the client wishes to update M to M' where $M' = M\Delta m$. To do so, the client can compute the tag t' for the updated

message M' just based on the update m and the state information about M that the client retained. Thus, the client can just upload the update m to the server and retain t' with it. We define the syntax and security of UdMACs and also propose two constructions for two different update functions. We prove the security of both constructions.

In Chapter 7, we use UdMACs to construct verifiable and fault-tolerant SSEs. In particular, in Chapter 7, we present a generic construction of a single keyword DSSE scheme that ensures forward privacy, backward privacy and correctness, even in the presence of a malicious server and a client issuing faulty updates. We refer to such a DSSE scheme as a *fault-tolerant verifiable dynamic SSE* or FVDSSE. Our construction combines UdMAC with a generic forward and backward private DSSE scheme that is secure in the honest-but-curious adversarial setting to develop an equivalently secure DSSE scheme that is secure against malicious adversaries in the presence of faulty updates. To the best of our knowledge, such a scheme is the first of its kind.

Chapter 8 summarizes our conclusions and discusses potential future directions for this area of research.

2 Preliminaries

In this chapter we discuss some preliminaries which would be used throughout this thesis. We begin with introducing some general notations in Section 2.1 followed by a discussion on the basic cryptographic objects and assumptions used in this thesis in Section 2.2. In Section 2.3, we discuss in detail the syntax of a dynamic SSE which is our main object of interest. We define important concepts related to conjunctive SSEs in Section 2.4. In Section 2.5, we define the security of SSE schemes, where we define the basic $\mathcal{L}$-adaptive security of SSEs and further discuss the other important dimensions of security of SSEs like correctness, soundness and fault tolerance. Section 2.5 also includes a detailed discussion about leakage functions which are fundamental to the security definitions of SSEs. In Section 2.6 we summarize some important previous works on SSEs.

2.1 General Notations

For a finite set X, $|X|$ denotes the cardinality of X. Let X be any set, then $x \xleftarrow{\$} X$ means x is sampled uniformly at random from the set X. Let X, Y be two finite sets then, $X \times Y$ denotes the Cartesian products of X and Y. For a non-negative integer n, $[n]$ denotes the set $\{1, 2, \ldots, n\}$, and for non-negative integers i, j, $i < j$, $[i, j]$ represents the set $\{i, i+1, ..., j\}$. The set $\{0,1\}^*$ represents the set of all binary strings, including the empty string, and for a positive integer n, $\{0,1\}^n$ denotes the set of all n bit strings. For $x, y \in \{0,1\}^*$, $x \| y$ denote the concatenation of the strings

x and y.

We sometimes see the set $\{0,1\}^n$ as the finite field $\mathbb{F}_{2^n}$ with 2^n elements. For such an interpretation, for $x, y \in \mathbb{F}_{2^n}$, we will denote addition and multiplication of x, y in $\mathbb{F}_{2^n}$ by $x \oplus y$ and xy respectively. We will explicitly mention, when we treat $\{0,1\}^n$ as $\mathbb{F}_{2^n}$.

We treat adversaries as algorithms. By a ppt adversary $\mathcal{A}$ we mean that the adversary runs in probabilistic polynomial time, i.e., $\mathcal{A}$ is a probabilistic algorithm which runs in polynomial time. Sometimes we describe adversaries with access to oracles. For an adversary $\mathcal{A}$, by $\mathcal{A}^{\mathcal{O}} = 1$, we mean that $\mathcal{A}$ has oracle access to $\mathcal{O}$, and after interacting with $\mathcal{O}$, $\mathcal{A}$ outputs a 1.

A function $f : \mathbb{N} \to \mathbb{R}$ is negligible if and only if for all $c > 0$, there exists a $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) < n^{-c}$. By $\mathsf{negl}(\lambda)$ we denote a negligible function in λ. For a natural number $n \in \mathbb{N}$, $\mathsf{poly}(n)$ denotes an arbitrary polynomial in n.

2.2 Hardness Assumptions

Pseudo-Random Function (PRF).

Let $\mathcal{K}$, $\mathcal{M}$, $\mathcal{R}$ be sets and let $F : \mathcal{K} \times \mathcal{M} \to \mathcal{R}$ be a two input function, which we sometimes see as a function family $F = \{F(K, \cdot)\}_{K \in \mathcal{K}}$. Let $\mathsf{Func}(\mathcal{M}, \mathcal{R})$ be the set of all functions mapping $\mathcal{M}$ to $\mathcal{R}$.

Definition 2.2.1 (PRF)**.** *Let $F : \{0,1\}^\lambda \times \mathcal{M} \to \mathcal{R}$ be an efficiently computable keyed function. F is said to be a pseudo-random function family if for all* ppt *adversary $\mathcal{A}$ the following holds.*

$$\mathsf{Adv}^{\mathsf{prf}}_{F,\mathcal{A}} = \left|\Pr\left[\mathcal{A}^{F_k(\cdot)}\left(1^\lambda\right)\right] - \Pr\left[\mathcal{A}^{\rho(\cdot)}\left(1^\lambda\right)\right]\right| \leq \mathsf{negl}(\lambda),$$

where probabilities are taken over random choices of $k \xleftarrow{\$} \{0,1\}^\lambda$ and $\rho \xleftarrow{\$} \mathsf{Func}(\mathcal{M}, \mathcal{R})$ and any randomness used by $\mathcal{A}$.

For denoting the PRF-advantage of an adversary $\mathcal{A}$ attacking a function family F we sometimes use the equivalent notation $\mathsf{Adv}^{\mathsf{prf}}_F(\mathcal{A})$.

Decisional Diffie-Hellman Assumption (ddh-Assumption) [29].

Let $\mathcal{G}$ be a ppt algorithm which on input λ outputs a description of a cyclic group G of prime order p, where $|p| = \mathsf{poly}(\lambda)$, and a generator $g \in G$. Then the *decisional Diffie-Hellman assumption* is the following.

Definition 2.2.2 (ddh-Assumption)**.** *We say that the* ddh *problem is hard relative to* $\mathcal{G}$ *if the following holds for all* ppt *adversary* $\mathcal{A}$*.*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ddh}}_{G,\mathcal{A}}(\lambda) &= \Big|\Pr\left[a,b \xleftarrow{\$} \mathbb{Z}_p^* : \mathcal{A}\left(g, g^a, g^b, g^{ab}\right) = 1\right] \\ &\qquad - \Pr\left[a,b,c \xleftarrow{\$} \mathbb{Z}_p^* : \mathcal{A}\left(g, g^a, g^b, g^c\right) = 1\right]\Big| \\ &\leq \mathsf{negl}(\lambda),\end{aligned}$$

where the probabilities are take over the randomness of $\mathcal{A}$ *and random choices of* $a, b, c \in \mathbb{Z}_p^*$*.*

Extended Decisional Diffie-Hellman Assumption (eddh-Assumption) [80].

Let $\mathcal{G}$ be a ppt algorithm which on input λ outputs a description of a cyclic group G of prime order p (st. $|p| = \mathsf{poly}(\lambda)$), and a generator $g \in G$. Then the *extended decisional Diffie-Hellman assumption* is the following.

Definition 2.2.3 (eddh-Assumption)**.** *We say that the* eddh *problem is hard relative to* $\mathcal{G}$ *if the following holds for all* ppt *adversary* $\mathcal{A}$*.*

$$\mathsf{Adv}^{\mathsf{eddh}}_{G,\mathcal{A}}(\lambda) = \Big|\Pr\left[\mathcal{A}\left(g, M\right) = 1\right] - \Pr\left[\mathcal{A}\left(g, \widehat{M}\right) = 1\right]\Big| \leq \mathsf{negl}(\lambda),$$

where

$$M = \begin{bmatrix} g^{\alpha_1\beta_1} & g^{\alpha_1\beta_2} & \cdots & g^{\alpha_1\beta_n} \\ g^{\alpha_2\beta_1} & g^{\alpha_2\beta_2} & \cdots & g^{\alpha_2\beta_n} \\ \vdots & \vdots & \ddots & \vdots \\ g^{\alpha_m\beta_1} & g^{\alpha_m\beta_2} & \cdots & g^{\alpha_m\beta_n} \end{bmatrix}, \widehat{M} = \begin{bmatrix} g^{\gamma_{1,1}} & g^{\gamma_{1,2}} & \cdots & g^{\gamma_{1,n}} \\ g^{\gamma_{2,1}} & g^{\gamma_{2,2}} & \cdots & g^{\gamma_{2,1}} \\ \vdots & \vdots & \ddots & \vdots \\ g^{\gamma_{m,1}} & g^{\gamma_{m,2}} & \cdots & g^{\gamma_{m,n}} \end{bmatrix}$$

the probabilities are take over the randomness of $\mathcal{A}$ *and for all* $i \in [m]$, $j \in [n]$, $\alpha_i, \beta_j, \gamma_{i,j} \xleftarrow{\$} \mathbb{Z}_p^*$.

2.3 Dynamic Searchable Symmetric Encryption

Let $\mathcal{D} = \{\mathsf{d}_1, \ldots, \mathsf{d}_D\}$ denote a collection of documents that the client wants to store in the server. Let $\mathcal{W}$ denote the set of all possible keywords. Each document $\mathsf{d}_i \in \mathcal{D}$, is composed of a set of keywords $\mathsf{w}_i \subseteq \mathcal{W}$. Define $\mathsf{W} = \bigcup_{i=1}^{D} \mathsf{w}_i$ to be the collection of all distinct keywords present in the database $\mathcal{D}$.

Let $\mathcal{I} = \{\mathsf{id}_1, \ldots, \mathsf{id}_D\} \subseteq \{0,1\}^\lambda$, for a fixed constant λ (in general considered as the security parameter), be the set of all identifiers. For each keyword, $w \in \mathsf{W}$, $\mathsf{db}(w)$ represents the set of identifiers of those documents containing the keyword w. For a $w \in \mathsf{W}$, let $t_w = \mathsf{db}(w) \times \{w\} = \{(\mathsf{id}, w) : \mathsf{id} \in \mathsf{db}(w)\}$. The database DB of an SSE scheme is viewed as the set of tuples

$$\mathsf{DB} = \bigcup_{w \in \mathcal{W}} t_w.$$

It is customary to abstract out a database as a collection of identifiers and their associated keywords as described above. This allows the decoupling of the storage of documents with the storage of the inverted index.

A DSSE scheme $\Lambda = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ is a three tuple, comprised of three protocols between the client (C) and the server (S). In the case where the SSE supports verification of the search result, the Search protocol of SSE is also equipped with another algorithm called Verify, run by the client. A static SSE scheme does not support the Update protocol. A protocol P between the client C and the server S, is denoted by $(\cdots \ ; \ \cdots) \leftarrow \mathsf{P}(\cdots \ ; \ \cdots)$. The semicolon separates the inputs and outputs of both parties. The first part before the semicolon is for the client and the part after the semicolon is for the server. We consider only those as output which both the parties get after the complete execution of the protocol.

During the initiation of the SSE scheme, the client runs the setup to obtain an

encrypted database EDB. The server is provided with EDB. Thereafter, the client can run three types of queries: addition, search, and deleting (add, srch, and del respectively) on the EDB. An SSE is primarily concerned with the storage and retrieval of the index, and not how the actual documents are stored. An update query is *always considered to be successful*. For search queries, in an honest-but-curious adversarial model, the query is always considered to be successful. In a malicious adversarial model, the client either outputs the set of identifiers matching the query rule or "reject" the result.

With this abstraction of a database, a DSSE scheme Λ is defined as follows. It closely resembles the definitions of [25] and [38]. The definition of DSSE in Definition 2.3.1 captures an honest-but-curious adversary. We'll amend the definition while considering a malicious adversary in Section 2.5.3.

Definition 2.3.1 (DSSE)**.** *A DSSE scheme* $\Lambda = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *is a three tuple of protocols between the client* (C) *and the server* (S) *defined as follows.*

- $(k, \sigma_{\mathsf{C}}\ ;\ \mathsf{EDB}) \leftarrow \mathsf{Setup}(1^{\lambda}, \mathsf{DB}\ ;\ \perp)$: *is a protocol executed between the client and the server. It comprises the following two algorithms.*

  $(k, \sigma_{\mathsf{C}}, \mathsf{EDB}) \leftarrow \mathsf{Setup}_{\mathsf{C}}(1^{\lambda}, \mathsf{DB})$: *This is a probabilistic polynomial time algorithm run by the client that takes as input the security parameter* 1^{λ} *and the initial database* DB*. It outputs a key* k*, the client's state* σ_{C} *and the encrypted database* EDB*. The client stores the key* (k, σ_{C}) *securely, and the encrypted database* EDB *is sent to the server.*

  $\mathsf{EDB} \leftarrow \mathsf{Setup}_{\mathsf{S}}(\mathsf{EDB})$: *upon receiving* EDB *from the client, the server saves it.*

- $(\sigma_{\mathsf{C}}\ ;\ \mathsf{EDB}) \leftarrow \mathsf{Update}(k, \sigma_{\mathsf{C}}, \mathsf{op}, (\mathsf{id}, w)\ ;\ \mathsf{EDB})$: *is a protocol executed between the client and the server. It comprises the following two algorithms.*

  $(\sigma_{\mathsf{C}}, \mathsf{utk}) \leftarrow \mathsf{Update}_{\mathsf{C}}(k, \sigma_{\mathsf{C}}, \mathsf{op}, (\mathsf{id}, w))$: *This (possibly probabilistic) algorithm is run by the client. On input the key* k*, client's state* σ_{C}*, an update*

query with operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, and a keyword-identifier pair* (id, w) *the algorithm outputs an update token* utk *and client's updated state* σ_{C}*.*

$\mathsf{EDB} \leftarrow \mathsf{Update}_{\mathsf{S}}(\mathsf{utk}, \mathsf{EDB})$: *This is a deterministic algorithm run by the server that takes as input the encrypted database* EDB *and the update token* utk*. It then outputs an updated encrypted database* EDB*.*

- $(\sigma_{\mathsf{C}}, \mathsf{db}(q)\ ;\ \mathsf{EDB}) \leftarrow \mathsf{Search}(k, \sigma_{\mathsf{C}}, q\ ;\ \mathsf{EDB})$: *is a protocol executed between the client and the server. It comprises the following two algorithms.*

  $(\sigma_{\mathsf{C}}, \mathsf{stk}) \leftarrow \mathsf{Search}_{\mathsf{C}}(k, \sigma_{\mathsf{C}}, q)$: *This (possibly probabilistic) algorithm is run by the client. On input the key* k*, client's state* σ_{C} *and search query* q*, the algorithm outputs a search token* stk *and client's updated state* σ_{C}*.*

  $\mathsf{res} \leftarrow \mathsf{Search}_{\mathsf{S}}(\mathsf{stk}, \mathsf{EDB})$: *This is a deterministic algorithm run by the server. It takes as input the encrypted database* EDB *and the search token* stk*. It outputs* res *as the search results. The final search result* $\mathsf{db}(q)$ *for search query* q*, is computed from* res*.*

Remark. In the update phase, we always consider *atomic updates*, which means a single keyword-identifier pair is updated at a time. In case of a bulk update (e.g. adding a file with many keywords), we invoke the update protocol multiple times. However, all the update tokens in such cases, can be communicated together.

Types of SSE. Depending upon the rule of search query and the number of keyword(s) involved in it, an SSE scheme can be classified into many types. However, we only discuss those types which are relevant to this thesis.

1. **Single-keyword SSE** supports only a single keyword per search query and returns all the matching document identifiers which match the searched keyword w. We call such a query $q = w$ as the equality query.
2. **Conjunctive SSE** supports querying a set of keyword $\{w_1, w_2, \ldots, w_n\}$ in each search query. The scheme, as a search result, returns the set of all the document

identifiers containing all the keywords in the query. For a conjunctive query, we write $q = \omega = w_1 \wedge w_2 \wedge \ldots \wedge w_n$, and the search result as $\mathsf{db}(\omega)$.

2.4 Conjunctive Dynamic SSE

An SSE scheme may allow searches using multiple keywords and their conjunctions. Let $L = \{w_1, w_2, \ldots, w_n\} \subseteq \mathsf{W}$ be a subset of keywords and $\omega(L) = w_1 \wedge w_2 \wedge \cdots \wedge w_n$ be a conjunction of the elements of L. For every conjunction $\omega(L)$, we assume the existence of a least frequent keyword, w.l.o.g say w_1.

Definition 2.4.1 (Least Frequent Term in a Conjunction [29])**.** *In a conjunctive query $\omega(L)$, a keyword $w_i \in L$ is termed the least frequent term in the conjunction if, for every $w_j \in L$ such that $w_j \neq w_i$ and w_j appears at least in as many updates as w_i does.*

In simple terms, a least frequent keyword is a keyword in L, which has appeared in the least number of updates among all the other keywords in L.

Definition 2.4.2 (*s-term* and *x-term* [29])**.** *In a conjunctive query $\omega(L)$ the least frequent term (say w_1) of the query is referred to as the "s-term" and all $w_i \in \widetilde{L} = L \setminus \{w_1\}$ are referred to as the "x-terms".*

For convenience, we denote a conjunctive query $\omega(L)$ as ω.

For a conjunctive query $\omega = w_1 \wedge w_2 \wedge \cdots \wedge w_n$, the set of identifiers matching the search query ω is denoted by $\mathsf{db}(\omega)$. Formally, for a conjunctive query ω, we define

$$\mathsf{db}(\omega) = \bigcap_{i=1}^{n} \mathsf{db}(w_i).$$

2.5 Security of DSSE

The security of a DSSE scheme Λ is determined by a leakage profile $\mathcal{L}_\Lambda$. The leakage profile $\mathcal{L}_\Lambda = (\mathcal{L}_{\Lambda,\mathsf{Setup}}, \mathcal{L}_{\Lambda,\mathsf{Search}}, \mathcal{L}_{\Lambda,\mathsf{Update}})$ denote the information that the adversary learns from the setup process and each execution of the search and update protocols.

$\mathcal{L}_{\Lambda,\mathsf{Setup}}$ denotes the leakage due to Setup. We note that the adversary only receives EDB_0 (initial encrypted database) from the output of $\mathsf{Setup}_\mathsf{C}$, where DB_0 is the initial database. So, $\mathcal{L}_{\Lambda,\mathsf{Setup}}$ is effectively the leakage from EDB_0. $\mathcal{L}_{\Lambda,\mathsf{Search}}$ denotes the leakage due to all the searches. $\mathcal{L}_{\Lambda,\mathsf{Update}}$ denotes the leakage due to all updates. The security of SSE schemes is generally argued by showing that an adversary cannot distinguish between the transcript of a real-world execution and an ideal-world execution (simulated using the leakage profile) of the scheme [42, 60, 28]. To formally define the security of SSE we introduce some more notations.

Query Sequence. A ℓ-query sequence QS_ℓ of a DSSE scheme Λ is a set with ℓ element, where $1 \leq \ell \leq \mathsf{poly}(\lambda)$ is the total number of queries. Formally, $\mathsf{QS}_\ell = \{Q_1, \ldots, Q_i, \ldots, Q_\ell\}$, is the list of all queries made by the client. Every element $Q_i = (u_i, \mathsf{op}_i, \mathsf{in}_i)$ of a query sequence is a three tuple, where u_i is the time-stamp of the query, $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{srch}, \mathsf{del}\}$ is the type/operation of the query, and in_i is the input of the query depending upon the operation op_i. If $\mathsf{op}_i = \mathsf{srch}$ is a search query, then $\mathsf{in}_i = q$[1]; if $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{del}\}$ is an update query, then $\mathsf{in}_i = (\mathsf{id}_i, w_i)$.

View of the Server. To define the view of the server in an SSE scheme, we use the formalization of [88, 38]. The "*view*" of the server in a protocol P is denoted by $\mathsf{T}[\mathsf{P}]$. For a DSSE scheme Λ, the first protocol invoked is always the Setup protocol. Then, for a query sequence QS_ℓ and for every query $Q_i \in \mathsf{QS}_\ell$ a protocol P_i is invoked, where $1 \leq i \leq \ell$. $\mathsf{P}_i = \mathsf{Search}$ if $\mathsf{op}_i = \mathsf{srch}$ and $\mathsf{P}_i = \mathsf{Update}$ if $\mathsf{op}_i \in \{\mathsf{add}, \mathsf{del}\}$. The view of the server in every individual protocol is formally defined as follows,

$$\mathsf{T}[\mathsf{Setup}] = \tau_0 = \emptyset, \qquad \mathsf{T}[\mathsf{P}_i] = \tau_i = \begin{cases} (u_i, \mathsf{stk}_i) & \text{if } \mathsf{P}_i = \mathsf{Search} \\ (u_i, \mathsf{utk}_i) & \text{if } \mathsf{P}_i = \mathsf{Update}. \end{cases}$$

for some $1 \leq i \leq \ell$. For every update query, the server gets an update token utk and for every search query, the server gets a search token stk along with their respective time stamps. So, τ_i is either the (u_i, utk_i) or (u_i, stk_i), depending upon whether the

[1]For example $q = w$ in case of a single keyword search query, and $q = \omega$ for conjunctive query.

query is an update query or a search query, respectively.

A "*transcript*" of a query sequence only contains the messages sent from the client for a search or an update query. The output of the server can efficiently be computed from the EDB and the search or update token sent by the client. This is because both the $\mathsf{Search}_\mathsf{S}$ and the $\mathsf{Update}_\mathsf{S}$ algorithms of Definition 2.3.1 are deterministic. With this, we finally define a transcript of a i-query sequence QS_i as follows.

$$\mathsf{Trans}_i[\mathsf{QS}_i] = \{\tau_j : 0 \leq j \leq i\},$$

for $1 \leq i \leq \ell$. For convenience, we write $\mathsf{Trans}_i[\mathsf{QS}_i]$ as Trans_i.

2.5.1 Security Game

The security of SSE is generally defined in a real-ideal simulation paradigm [42, 60, 28]. A leakage profile $\mathcal{L}_\Lambda$ is defined to capture all the leakages (due to the setup, search and update protocols). A simulator is given access to the leakage function to simulate the real-world execution of the scheme. Thus, the security definition is parameterised by the leakage profile.

Definition 2.5.1 (Adaptive security of DSSE)**.** *Let* $\Lambda = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a DSSE scheme and* $\mathcal{L}_\Lambda = (\mathcal{L}_{\Lambda,\mathsf{Setup}}, \mathcal{L}_{\Lambda,\mathsf{Search}}, \mathcal{L}_{\Lambda,\mathsf{Update}})$ *be a triple of functions. Then, for any* ppt *adversary* $\mathcal{A}$ *and simulator* $\mathcal{S}$ *with the leakage function* $\mathcal{L}_\Lambda$*, we define the experiment* $\mathsf{SSEReal}^\Lambda_\mathcal{A}(\lambda)$*, and* $\mathsf{SSEIdeal}^\Lambda_{\mathcal{A},\mathcal{S}}(\lambda)$ *as follows.*

$\underline{\mathsf{SSEReal}^\Lambda_\mathcal{A}(\lambda):}$ *The adversary* $\mathcal{A}(1^\lambda)$ *chooses a* DB *and the challenger runs* $(\sigma_\mathsf{C},\ \mathsf{EDB}) \leftarrow \mathsf{Setup}_\mathsf{C}\left(1^\lambda, \mathsf{DB}\right)$ *and returns* EDB *to* $\mathcal{A}$*. Then for subsequent search or update queries, the challenger runs* $(\sigma_\mathsf{C}, \mathsf{stk}) \leftarrow \mathsf{Search}_\mathsf{C}(\sigma_\mathsf{C}, q)$ *or* $(\sigma_\mathsf{C}, \mathsf{utk}) \leftarrow \mathsf{Update}_\mathsf{C}(\sigma_\mathsf{C}, \mathsf{op}, (\mathsf{id}, w))$ *respectively and provides the adversary with* stk *or* utk*. Finally, after the search/update queries, the adversary* $\mathcal{A}$ *stops by outputting a bit, which is the output of the experiment.*

$\underline{\mathsf{SSEIdeal}^\Lambda_{\mathcal{A},\mathcal{S}}(\lambda):}$ *The adversary* $\mathcal{A}\left(1^\lambda\right)$ *chooses a* DB *and the challenger runs* $\mathsf{EDB} \leftarrow \mathcal{S}\left(\mathcal{L}_{\Lambda,\mathsf{Setup}}\right)$ *and returns* EDB *to* $\mathcal{A}$*. Then for subsequent search or update queries, the*

challenger runs $\mathsf{stk} \leftarrow \mathcal{S}(\mathcal{L}_{\Lambda,\mathsf{Search}})$ *or* $\mathsf{utk} \leftarrow \mathcal{S}(\mathcal{L}_{\Lambda,\mathsf{Update}})$ *respectively and provides the adversary with* stk *or* utk. *Finally, after the search/update queries, the adversary* $\mathcal{A}$ *stops by outputting a bit, which is the output of the experiment.*

A DSSE scheme Λ *is said to be* $\mathcal{L}_\Lambda$*-adaptively secure if for all adversaries* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that, the SSE advantage* $\left(\mathsf{Adv}^{\mathsf{Priv},\Lambda}_{\mathcal{A},\mathcal{S}}(\lambda)\right)$ *of* $\mathcal{A}$ *(defined below) is negligible in* λ.

$$\mathsf{Adv}^{\mathsf{Priv},\Lambda}_{\mathcal{A},\mathcal{S}}(\lambda) = \left|\Pr\left[\mathsf{SSEReal}^{\Lambda}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Lambda}_{\mathcal{A},\mathcal{S}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

2.5.2 Correctness of DSSE

A DSSE scheme Λ is *correct*, informally, if the result returned by the search protocol for a search query q is the list of document identifiers matching the query, i.e., $(\mathsf{db}(q))$, except with negligible probability. To define the correctness of a DSSE scheme we use the formalization provided in [28, 38]. We assume that the adversary makes ℓ-many queries where $\ell \leq \mathsf{poly}(\lambda)$. The correctness game of a DSSE is defined in Figure 2-1.

$\mathsf{DSSECorr}^{\Lambda}_{\mathcal{A}}(\lambda)$:

01. $\mathsf{flag} \leftarrow 0$
02. $(\mathsf{DB}, \sigma_{\mathcal{A}}) \leftarrow \mathcal{A}(1^\lambda)$
03. $(\sigma_{\mathsf{C}0}, \mathsf{EDB}_0) \leftarrow \mathsf{Setup}(\mathsf{DB})$
04. **for** $1 \leq i \leq \ell$ **do**
05. $\quad(\sigma_{\mathcal{A}}, q_i) \leftarrow \mathcal{A}(\sigma_{\mathcal{A}}, \mathsf{EDB}_0, \mathsf{Trans}_{i-1}) \quad \triangleright\ q_i = (\mathsf{op}_i, \mathsf{in}_i)$
06. $\quad$**if** $\mathsf{op}_i = \mathsf{srch}$
07. $\quad\quad(\sigma_{\mathsf{C}i}, \mathsf{res}_i\ ;\ \mathsf{EDB}_i) \leftarrow \mathsf{Search}(\sigma_{\mathsf{C}i-1}, q_i\ ;\ \mathsf{EDB}_{i-1})$
08. $\quad\quad\tau_i \leftarrow \mathsf{T}[\mathsf{Search}(\sigma_{\mathsf{C}i-1}, q_i\ ;\ \mathsf{EDB}_{i-1})]$
09. $\quad\quad$**if** $\mathsf{res}_i \neq \mathsf{db}(q_i)$
10. $\quad\quad\quad\mathsf{flag} \leftarrow 1$
11. $\quad\quad$**end if**
11. $\quad$**else**
12. $\quad\quad(\sigma_{\mathsf{C}i}, \mathsf{EDB}_i) \leftarrow \mathsf{Update}(\sigma_{\mathsf{C}i-1}, q_i; \mathsf{utk}_i, \mathsf{EDB}_{i-1})$
13. $\quad\quad\tau_i \leftarrow \mathsf{T}[\mathsf{Update}(\sigma_{\mathsf{C}i-1}, q_i; \mathsf{utk}_i, \mathsf{EDB}_{i-1})]$
14. $\quad$**end if**
15. **end for**
16. **return** flag

Figure 2-1: Correctness Game of DSSE.

In the correctness game, the adversary submits an initial database DB_0, to which it receives an encrypted database EDB_0. The adversary then makes repeated search or update queries to which it receives a transcript. The DSSE scheme Λ is *correct* if the result returned by the search protocol is *correct* i.e., the search result is $\mathsf{db}(q)$, except with negligible probability.

Definition 2.5.2 (Correctness of DSSE)**.** *Let,* $\Lambda = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a DSSE scheme, we say* Λ *is correct if for all* ppt *adversary* $\mathcal{A}$*, following holds.*

$$\mathsf{Adv}^{\mathsf{Corr},\Lambda}_{\mathcal{A}}(\lambda) \;=\; \Pr[\mathsf{DSSECorr}^{\Lambda}_{\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the DSSECorr *game is defined in Figure 2-1.*

2.5.3 Soundness of DSSE

In Section 2.3, the Definition of SSE (Definition 2.3.1) is presented assuming an honest-but-curious adversary. Where the adversary always follows the protocol correctly and replies with the correct search result. However, the adversary can be malicious, where it differs from the protocol to save storage and/or CPU time and tries to forge the client with incorrect search results. In such cases, the client needs a guarantee from the SSE protocol that the search result is correct.

A verifiable DSSE (VDSSE) ensures that the search result returned by the server is indeed correct. In a VDSSE the Search protocol is equipped with another additional algorithm called Verify. And the output of the search protocol of the client is either $\mathsf{db}(q)$ or "reject". The search protocol of a VDSSE is described as follows. All other protocols (i.e., Setup and Update) are the same as Definition 2.3.1.

$(\sigma_{\mathsf{C}}, (\mathsf{db}(q)$ **or** $\mathsf{reject})\ ;\ \mathsf{EDB}) \leftarrow \mathsf{Search}(k, \sigma_{\mathsf{C}}, q\ ;\ \mathsf{EDB})$: is a protocol executed between the client and the server. It comprises the following two algorithms.

$(\sigma_{\mathsf{C}}, \mathsf{stk}) \leftarrow \mathsf{Search}_{\mathsf{C}}(k, \sigma_{\mathsf{C}}, q)$: this (possibly probabilistic) algorithm is run by the client. On input the key k, client's state σ_{C} and search query q, the algorithm outputs a search token stk and client's updated state σ_{C}.

$\mathsf{res} \leftarrow \mathsf{Search}_{\mathsf{S}}(\mathsf{stk}, \mathsf{EDB})$: this is a deterministic algorithm run by the server. It takes as input the encrypted database EDB and the search token stk. It outputs a set res as the search results $\mathsf{db}(q)$ for search query q which is returned to the client.

The client then passes the search result res through another algorithm called Verify, described as follows.

$(\mathsf{db}(q)$ **or** $\mathsf{reject}) \leftarrow \mathsf{Verify}(k, \sigma_{\mathsf{C}}, \mathsf{res})$: this is a deterministic algorithm which on input the secret key k, the state σ_{C} and the result res returned by the server, outputs either the correct search result $\mathsf{db}(q)$ for the query q or "reject".

$\mathsf{DSSESound}^{\Lambda}_{\mathcal{A}}(\lambda)$:

```
01. flag ← 0
02. (σ_A, DB) ← A(1^λ)
03. (σ_C0, EDB_0) ← Setup(DB_0)
04. for 1 ≤ i ≤ ℓ do
05.    (σ_A, q_i) ← A(σ_A, EDB_0, Trans_{i-1})     ▷ q_i = (op_i, in_i)
06.    if op_i = srch
07.       (σ_Ci, res_i ; EDB_i) ← Search(σ_C{i-1}, q_i ; EDB_{i-1})
08.       τ_i ← T[Search(σ_C{i-1}, q_i ; EDB_{i-1})]
09.       if Verify(res_i) ≠ reject and res_i ≠ db(q_i)
10.          flag ← 1
11.       end if
11.    else
12.       (σ_Ci, EDB_i) ← Update(σ_C{i-1}, q_i; utk_i, EDB_{i-1})
13.       τ_i ← T[Update(σ_C{i-1}, q_i; utk_i, EDB_{i-1})]
14.    end if
15. end for
16. return flag
```

Figure 2-2: Soundness Game of DSSE

The soundness game of a DSSE is defined in Figure 2-2. In the soundness game, the adversary submits an initial database DB_0, to which it receives an encrypted database EDB_0. The adversary then makes repeated search or update queries to which it receives a transcript. Informally we say a DSSE scheme Λ is *sound* if the result returned by the search protocol is not rejected and is *correct* i.e., the search

result is either $\mathsf{db}(q)$ or "reject". To define the soundness of a DSSE scheme, we use the formalization provided in [24].

Definition 2.5.3 (Soundness of DSSE)**.** *Let,* $\Lambda = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *be a DSSE scheme, we say* Λ *is sound if for all* ppt *adversary* $\mathcal{A}$*, following holds.*

$$\mathsf{Adv}^{\mathsf{Sound},\Lambda}_{\mathcal{A}}(\lambda) \;=\; \Pr[\mathsf{DSSESound}^{\Lambda}_{\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the DSSESound *game is defined in Figure 2-2.*

2.5.4 Fault-tolerant Verifiable DSSE

A DSSE scheme is said to be fault-tolerant and verifiable DSSE if it archives correctness and soundness according to Definition 2.5.2 and 2.5.3 respectively even in the presence of incorrect updates, defined as follows.

Definition 2.5.4 (Incorrect Update)**.** *An update operation* $(\mathsf{op}, (\mathsf{id}, w))$ *is said to be incorrect if any of the following holds*

1. *if* $\mathsf{op} = \mathsf{del}$ *and* $\mathsf{id} \notin \mathsf{db}(w)$
2. *if* $\mathsf{op} = \mathsf{add}$ *and* $\mathsf{id} \in \mathsf{db}(w)$.

Remark. If a DSSE is not fault-tolerant, then the adversary in the correctness and the soundness game of Definition 2.5.2 and 2.5.3 respectively are not allowed to make any incorrect update according to Definition 2.5.4.

2.5.5 Forward and Backward Privacy (Single Keyword DSSE)

The two most essential security requirements (other than adaptive security) for a DSSE scheme are *forward privacy* and *backward privacy*. Intuitively, forward private SSE schemes ensure that future update queries do not leak any information about previous search operations performed on the inverted index. Similarly, backward privacy ensures that after a pair (id, w) has been deleted, the identifier id will not be

leaked in future search operations for w. Depending upon the information they leak during search and update operations, various flavours of backward privacy have been defined in the literature [25]. They are:

1. **Backward privacy with insertion pattern (BPIP):** This notion of backward privacy leaks the document identifiers currently matching the queried keyword w, the time when they were inserted into the encrypted inverted index, and the total number of updates performed on the searched keyword w. This security notion is popularly referred to in the literature as Type I backward privacy.
2. **Backward privacy with update pattern (BPUP):** This notion of backward privacy in addition to the BPIP leakages above, also leaks the time when all the updates on the queried keyword w happened without leaking their specific content. This security notion is popularly referred to in the literature as Type II backward privacy.
3. **Weak backward privacy (WBP)**: This notion of backward privacy in addition to the leakages in the previous two leakages BPIP and BPUP, also leaks the deletion updates that removed corresponding insertion updates. This security notion is popularly referred to in the literature as Type III backward privacy.

Some Standard Leakage Functions.

To define forward and backward privacy, we need some auxiliary definitions, which we introduce here.

$\mathsf{sp}(w)$: The search pattern leakage $\mathsf{sp}(w)$ in terms of SSE refers to the information of those time stamps when the keyword has been searched. The searched pattern for a keyword w is the set of timestamps of all search queries with the keyword w. Formally,

$$\mathsf{sp}(w) = \{u : (u, \mathsf{srch}, w) \in \mathsf{QS}\}.$$

$\mathsf{Hist}(w)$: This leakage function captures all the modifications that were made to the set $\mathsf{db}(w)$. This is the combination of two leakages, the leakage from $\mathsf{db}_0(w)$ (the set of document identifiers matching w at the setup) and the set $\mathsf{UpHist}(w)$, defined

below.

$$\mathsf{UpHist}(w) = \{(u, \mathsf{op}, \mathsf{id}) : (u, \mathsf{op}, (\mathsf{id}, w)) \in \mathsf{QS} \wedge \mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}\}.$$

$\mathsf{TimeDB}(w)$: This leakage function is defined as the timestamped list of documents currently matching the queried keyword w. We consider the refined definition provided in [38] to correctly capture the timestamps of entries that have not been deleted yet.

$$\mathsf{TimeDB}(w) = \{(u, \mathsf{id}) : ((u, \mathsf{add}, (\mathsf{id}, w)) \in \mathsf{QS}) \wedge (\forall u' > u, (u', \mathsf{del}, (\mathsf{id}, w) \notin \mathsf{QS}))\}.$$

$\mathsf{Updates}(w)$: Similarly, we define another leakage function $\mathsf{Updates}(w)$ as the set of timestamps of all update queries for a given w as,

$$\mathsf{Updates}(w) = \{u : (u, \mathsf{op}, (\mathsf{id}, w)) \in \mathsf{QS}\}.$$

$\mathsf{DelHist}(w)$: Finally, we define $\mathsf{DelHist}(w)$ as the set of all pairs of timestamps of addition and deletion for a given w.

$$\mathsf{DelHist}(w) = \left\{\left(u^{\mathsf{add}}, u^{\mathsf{del}}\right) : \left(\left(u^{\mathsf{add}}, \mathsf{add}, (\mathsf{id}, w)\right) \in \mathsf{QS}\right) \wedge \left(\left(u^{\mathsf{del}}, \mathsf{del}, (\mathsf{id}, w)\right) \in \mathsf{QS}\right)\right\}.$$

With the above definitions in place, we are now ready to provide the definition of forward and backward privacy for a $\mathcal{L}_\Lambda$-adaptively secure single keyword SSE Λ. The definition that we present is primarily the one presented in [25] with some modifications as suggested in [38]. Before we give the definition we discuss some history of the evolution of the definitions of forward and backward security.

The notion of forward and backward privacy was first introduced in [88]. However, forward privacy was first formalised in [23]. The first formal notion of backward privacy for single keyword DSSE was introduced in [25]. The authors of [38] modified the definition of backward privacy in [25] in several ways. The primary observations of [38] regarding the definition in [25] are the following:

- The definition in [25] does not capture a specific type of leakage called *search pattern* leakage correctly.
- The definition in [25] considers that an identifier-keyword pair, once inserted and then deleted, cannot be reinserted again into the database under the *weak backward privacy* (WBP) security model. This restriction in the context of WBP security model is artificial.
- Also, the security definition of WBP in [25] missed the leakage component $\mathsf{Updates}(w)$. Authors of [38] showed that the leakage $\mathsf{Updates}(w)$ cannot be derived from the definition of WBP provided in [25], thus, should be included explicitly in the definition of WBP

In [38] the definition in [25] was modified to take care of all the above points, i.e., the definition in [38] takes care of search pattern leakages and also lifts the artificial restriction for the WBP security model. In doing so, the authors of [38] provided an alternate definition of backward privacy called, *Backward Privacy with Link Pattern* (BPLP). We require some extra leakage function to define BPLP. However, this alternate formulation of backward privacy is not explicitly required for the works presented in this thesis. Therefore, we adopt the definitions of forward and backward privacy from [25] with the modification suggested in [38].

We will need some additional notations. For two leakage functions $\mathcal{L}_1$ and $\mathcal{L}_2$, $\mathcal{L}_1 \preceq \mathcal{L}_2$ denotes that "$\mathcal{L}_1$ *leaks less than* $\mathcal{L}_2$". In other words, $\mathcal{L}_1$ *leaks less than* $\mathcal{L}_2$ means that leakage information given by $\mathcal{L}_1$ about the database is less than $\mathcal{L}_2$, or, equivalently, all the information that can be inferred from $\mathcal{L}_1$ can be inferred from $\mathcal{L}_2$. Whenever, $\mathcal{L}_1$ leaks strictly less than $\mathcal{L}_2$, we write $\mathcal{L}_1 \prec \mathcal{L}_2$.

It is a usual practice to assume the initial database of any SSE to be empty. Thus, the leakage of the setup algorithm is null. However, if the initial database is not empty, then $\mathcal{L}_{\Lambda,\mathsf{Setup}}(\mathsf{DB}) = N$, where N denotes the number of keyword document pairs in the initial database. The definition of forward and backward privacy of SSE that we present next assumes the initial database to be empty.

Definition 2.5.5 (Forward Privacy of Single keyword DSSE)**.** *An* $\mathcal{L}_\Lambda = (\mathcal{L}_{\Lambda,\mathsf{Setup}},$

$\mathcal{L}_{\Lambda,\mathsf{Search}}$, $\mathcal{L}_{\Lambda,\mathsf{Update}}$)-*adaptive secure DSSE scheme* Λ *is forward private if its leakage function* $\mathcal{L}_\Lambda$ *can be written as follows.*

$$\mathcal{L}_{\Lambda,\mathsf{Setup}}(\mathsf{DB}) = \emptyset, \qquad \mathcal{L}_{\Lambda,\mathsf{Search}}(w) \preceq \{\mathsf{sp}(w), \mathsf{UpHist}(w)\},$$
$$\mathcal{L}_{\Lambda,\mathsf{Update}}(\mathsf{op}, \mathsf{in}) \preceq \{\mathsf{op}\}.$$

Definition 2.5.6 (Backward Privacy of Single keyword SSE)**.** *An* $\mathcal{L}_\Lambda = (\mathcal{L}_{\Lambda,\mathsf{Setup}}$, $\mathcal{L}_{\Lambda,\mathsf{Search}}$, $\mathcal{L}_{\Lambda,\mathsf{Update}})$*-adaptive secure DSSE scheme* Λ *is backward private as per the notions* BPIP, BPUP *and* WBP *if its leakage functions* $\mathcal{L}_\Lambda$ *can be written as the follows.*

$$\begin{aligned}
\mathcal{L}^{\mathsf{BPIP}}_{\Lambda,\mathsf{Setup}}(\mathsf{DB}) &= \emptyset, \qquad \mathcal{L}^{\mathsf{BPIP}}_{\Lambda,\mathsf{Update}}(\mathsf{op}, \mathsf{in}) \preceq \{\mathsf{op}\}, \\
\mathcal{L}^{\mathsf{BPIP}}_{\Lambda,\mathsf{Search}}(w) &\preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), |\mathsf{Updates}(w)|\}, \\
\mathcal{L}^{\mathsf{BPUP}}_{\Lambda,\mathsf{Setup}}(\mathsf{DB}) &= \emptyset, \qquad \mathcal{L}^{\mathsf{BPUP}}_{\Lambda,\mathsf{Update}}(\mathsf{op}, \mathsf{in}) \preceq \{\mathsf{op}, w\}, \\
\mathcal{L}^{\mathsf{BPUP}}_{\Lambda,\mathsf{Search}}(w) &\preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w)\}, \\
\mathcal{L}^{\mathsf{WBP}}_{\Lambda,\mathsf{Setup}}(\mathsf{DB}) &= \emptyset, \qquad \mathcal{L}^{\mathsf{WBP}}_{\Lambda,\mathsf{Update}}(\mathsf{op}, \mathsf{in}) \preceq \{\mathsf{op}, w\}, \\
\mathcal{L}^{\mathsf{WBP}}_{\Lambda,\mathsf{Search}}(w) &\preceq \{\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{Updates}(w), \mathsf{DelHist}(w)\}.
\end{aligned}$$

In our work, specifically in Chapter 5, we are required to define forward and backward privacy for conjunctive SSEs. We modify the above definitions, which are essentially the definitions presented in [38], to make them suitable for conjunctive SSEs.

2.6 Related Works

The Early Schemes. SSE was first introduced in [86]. One of the main drawbacks of the scheme in [86] was that the search time was linear in the number of documents. This limitations were lifted in the works reported in [50] and [34]. Constructions based on an inverted index were first proposed in [50]. The use of an inverted index was significant, as its use ensured that the search time was proportional to the

number of documents matching the keyword in the collection. Authors of [50] first introduced formal security notions of SSE. They proposed the notion of indistinguishability against chosen keyword attacks (IND-CKA) and the slightly stronger IND2-CKA) notion for the security of indexes, they also provide a construction based on Bloom filters [17] and pseudo-random functions. Authors of [34] provide a construction that achieves security similar to IND2-CKA.

The first strong notion of security, which is still in use, was first proposed in [42]. However, the scheme described in [42] was only *static*. The first truly dynamic SSE with search complexity $\mathcal{O}(\mathsf{db}(w))$ was proposed in [60]. Since then, several dynamic single keyword SSE schemes have been proposed.

Attacks on SSE. User queries do leak some information to the server regarding searches and updates. Determining such leakages is important. There have been several works which describe attacks on SSE schemes which utilize the leakages during search and updates [55, 68, 27]. A unifying theme of these attacks was to use knowledge of the plain text database apriori. One of such attacks reported in [97] extended the attack by [27] to demonstrate that the server may inject only a few documents (through the client) into the database, to be able to recover a large fraction of the keywords searched by the client. Authors of [1] showed the consequences of this attack on relational databases.

Forward and Backward Privacy. The attacks of [97] above created a necessity to update the security definition of SSEs. To mitigate such attacks, two new notions of privacy for DSSE, namely, *"forward privacy"* and *"backward privacy"*, were first proposed in [88]. Intuitively, forward private SSE schemes ensure that the server cannot link an update operation for (id, w) with previous search operations performed on the inverted index with w or containing id in its result. This protects queries made to the inverted index with newly injected files by the adversary as was used in the attacks mounted in [97]. Backward privacy ensures that after a pair (id, w) has been deleted, the identifier id will not be leaked in future searches for w.

The first formal definition and a scheme achieving forward privacy were proposed in [23]. The scheme of [23] uses a trapdoor one-way permutation to achieve forward privacy. The first formal definition of backward privacy was defined in [25]. Depending upon the information leaked during search and update operations, various flavours of backward privacy have been defined in the literature [25], which has been followed in subsequent works [49, 90, 101, 89]. Authors of [25] defined three types of backward privacy, namely, BPIP, BPUP and WBP in increasing order of leakage (See Definition 2.5.6) to the server (decreasing order of security). These three types are popularly referred to as Type I, Type II and Type III backward privacy in literature. The authors of [101] gave a construction using a bit-map index with homomorphic addition to achieve Type I$^-$ security whose leakage is even lesser than a Type I secure scheme. However, their storage and search complexities are high at $\mathcal{O}(u|D|)$, where u is the number of updates for a keyword. More efficient construction of a Type III backward private SSE scheme was proposed in [89] that uses *symmetric puncturable encryption*. Since then, several forward and backward private SSE schemes [49, 101, 38] have been proposed, improving security and efficiency.

In [38], the authors demonstrated that the definition of backward privacy proposed in [25] was incomplete, and it didn't cover all general scenarios. Also, the authors of [25] did miss some leakage while defining backward privacy. In [25], a relaxed version of SSE was considered in their Type III security definition. They considered that once a keyword-identifier pair is added and then deleted, the same pair cannot be re-inserted into the database. This restriction was lifted in [38] to provide a more general and complete definition of backward privacy for SSE with all perceivable and known leakages. Although there are no known practical attacks on SSE schemes due to a lack of backward privacy, it provides meaningful theoretical assurance.

SSE Supporting Complex Queries. Most SSE schemes [42, 60, 28, 23, 87, 25, 49, 89, 38, 90] allow searching for a single keyword in each query. Supporting conjunctive queries and/or general boolean queries with efficient computation and communication along with adequate security (low leakage) is a well-known challenge in the SSE

literature. There are a few SSE schemes that support conjunctive queries [29, 47, 2, 93, 58, 66, 80, 102, 100, 95]. The most notable of these is the oblivious cross-tag (OXT) system from Crypto'13 [29]. It assumes that the client knows the *least frequent keyword* w_1 in a conjunction $w_1 \wedge \cdots \wedge w_n$. This keyword w_1 is called the *s-term* and all other keywords $w_2, \ldots, w_n$ are called *x-terms* [29]. OXT uses two data structures T-Set and X-Set. During a search operation in OXT, the server first conducts a single keyword search using the T-Set to find all ids containing the *s-term* w_1. For all such ids containing w_1, the X-Set is used to check if the *x-terms* $w_i, i \geq 2$ are also in id. The OXT scheme is quite efficient with its search complexity of $\mathcal{O}(n \cdot |\mathsf{db}(w_1)|)$, where n is the number of keywords (*s-term* and *x-terms*) in the conjunction and the set $\mathsf{db}(w_1)$ contains the ids of documents where the *s-term* w_1 appears.

The main criticism of the OXT scheme was that besides being a static scheme, it leaked keyword pair result patterns (KPRP). Hiding KPRP is important in the context of the attacks presented in [97]. Moreover, OXT uses exponentiation in ddh-hard groups, which is a costly operation and removing this would lead to making OXT more efficient. Subsequently, proposed static schemes like [66, 78] were directed towards mending these shortcomings of OXT. The work in [66] used significant pre-processing on the data to restrict KPRP leakage. Their construction uses Bloom filters and symmetric hidden vector encryption in a novel way. The construction of [78] has designed a novel data structure called ConjFiler, which allows searching for conjunction only using symmetric key primitives. However, ConjFiler requires to pre-compute all possible two-conjunction to facilitate conjunctive query. This is why making ConjFiler dynamic is a challenging task.

The first proposed construction of a forward and backward secure CDSSE was in [80] through a new construction called Oblivious Dynamic Cross Tags (ODXT), that successfully managed the pre-processing of OXT in the dynamic setting. However, the attack in [100] showed that the construction in [80] failed to provide forward privacy in some cases. Few other forward and backward private CDSSE schemes [102, 100, 95] have followed, achieving forward privacy and different flavours of backward privacy with efficiency trade-offs. However, these schemes are far from achieving the

efficiency of the OXT scheme. The scheme of [102] performs $\mathcal{O}(|D|)$ computation for search, where $|D|$ is the number of documents in the database, making it prohibitively slow. The scheme of [100] uses a single keyword SSE (SKSSE) to store and fetch the elements in the conjunction which makes it very slow as well. The scheme of [95] runs in sub-linear time but is still far from the practical performance of OXT. In [95], the computation and communication complexities of the search operation are both $\mathcal{O}\left(u_1 + n \cdot \mathsf{db}(w_1)\right)$ and involves two rounds, where u_1 is the number of updates related to w_1; the complexities of the update operation are $\mathcal{O}\left(|W|/|W_d|\right)$ for adding a document, $\mathcal{O}\left(|D|\right)$ for editing and $\mathcal{O}\left(1\right)$ for deleting, where $|W|$ is the number of keywords in the database and $|W_d|$ denotes the number of keywords contained in the updated document. The storage requirement of [95] is $\mathcal{O}(|W||D|)$, as compared to $\mathcal{O}(N)$ for our scheme, where N is the number of identifier-keyword pairs in the database. Typically, N is much smaller than $|W||D|$ in practice. There is no additional storage requirement due to the keyword updates in [95].

Verifiable SSE. In SSE, it's assumed the server (seen as an adversary) is honest-but-curious. However, the servers can be "malicious". Verifiable SSE (VSSE) provides cryptographic guarantees against such servers.

The first universally composable secure VSSE scheme was introduced in [63] against non-adaptive adversaries. The first adaptive secure VSSE was proposed in [30], though both works addressed the static SSE setting. Authors of [64] extended their previous work [63] to a verifiable dynamic SSE (VDSSE), supporting updates operation, but it lacked forward privacy. The first forward private VDSSE was proposed in [24], using the incremental multi-set hashing proposed in [40]. Subsequently, Zhang et al. [98] introduced another incremental multi-set hash-based SSE using symmetric key primitives. The client-side storage required for both schemes was high. Authors of [48] addressed this issue by proposing a novel data structure called the *Accumulative Authentication Tag* (AAT), which reduces storage overhead while using symmetric key primitives.

While all these constructions supported verification of the search result, almost all

of them failed to provide correct proof in case of faulty updates from the client. There has been limited research focused on designing DSSE schemes that are fault-tolerant, verifiable, and both forward and backward private. In [88], authors proposed a DSSE scheme using an ORAM-style data structure, but it required frequent rebuild operations. This scheme was forward secure and fault-tolerant, but not verifiable. Also, the search operation in this scheme is inefficient, taking linear time (on the number of documents) in the worst case. Authors of [24] proposed three forward private, verifiable DSSE schemes GVS-Hash, GVS-Acc and GVS-Acc-RSA. The first construction is based on Merkle tree-like data structures and Multi-Set Hash, while the latter two are based on cryptographic accumulators. However, none of these constructions are fault-tolerant and they do not scale well with large databases. In addition to these constructions, authors of [24] also proposed verifiable versions of Linear SPS and Sublinear SPS introduced in [88]. Recently, an efficient FVDSSE construction was proposed in [94] that employs *authenticated encryption* (AE) for verification, avoiding the expensive use of Merkle trees or accumulators, thereby making the construction very fast in practice. Their proposed generic scheme, when instantiated with a forward private scheme [87], ensures forward privacy. However, this construction fails to provide backward privacy.

2.7 Final Remarks

In this Chapter, we discussed the preliminaries required to appreciate the work reported in the subsequent chapters. We listed the general notations which are heavily used throughout the thesis, some less used notations would be introduced further wherever necessary. We also discussed all necessary background for SSE schemes including their syntax, security and variants. We also provided a comprehensive discussion on the related works which trace the history of the development of SSE. Though we do not pretend that our coverage of SSE schemes in this chapter is exhaustive, we briefly discuss all works which are of relevance to the work that we present in the following chapters.

3 Tree Covers

We already described in Section 2.3 that Searchable Symmetric Encryption schemes, as introduced in [86, 42], abstract a database (DB) as a set of keyword-identifier pairs. Formally,

$$\mathsf{DB} = \bigcup_{w \in \mathsf{W}} \{(\mathsf{id}, w) : \mathsf{id} \in \mathsf{db}(w)\},$$

where $\mathsf{db}(w)$ is the set of identifiers corresponding to the keyword w. An SSE encrypts this set, comprising keyword-identifier pairs, using a specialized structure often referred to as an "*encrypted multi-map*" [35] (EMM) or an inverted index. This encrypted multi-map is stored on the server and facilitates both searching and updating the encrypted database.

The efficiency of an SSE scheme is measured by the time it takes to search and update. Most schemes are optimized to get better search and update times maintaining adequate security guarantees. Optimizing the size of the encrypted database has not yet received adequate attention in the literature, as it is always assumed that the server is computationally powerful and has sufficient storage. But, optimizing the size of the EMM has an immediate effect on search time and communication overhead. If the input set of tuples for an SSE is DB, then the size of the encrypted database in any existing SSE scheme, to the best of our knowledge, is at least $|\mathsf{DB}|$. In fact, in most schemes (particularly in dynamic SSE schemes), the storage overhead is much more than $|\mathsf{DB}|$. Similarly, the size of the result of a query w is at least $|\mathsf{db}(w)|$, and in most schemes, the size of the search result which is to be communicated to the

client is much more than $|\mathsf{db}(w)|$. In most dynamic SSEs the size of the encrypted database and the query result increases with the number of updates performed on the database.

Recently, some effort has been made to restrict the size of the search result to at most $\mathcal{O}(\mathsf{db}(w))$ for any given keyword w [88, 49, 43, 33]. To the best of our knowledge, no SSE scheme has considered reducing the storage size beyond $|\mathsf{DB}|$ and the result size beyond $|\mathsf{db}(w)|$. In this study, we specifically address the following questions.

1. Can we build a functional and secure SSE whose storage size is smaller than $|S|$ on average?
2. Can a search query for a keyword w on average be answered with a result size smaller than $|\mathsf{db}(w)|$?

We answer both these questions in the affirmative. In order to achieve this, we introduce a generic method for transforming any single keyword SSE scheme into an equivalent secure SSE scheme where the size of the encrypted database required to store for searching is much smaller than in the original SSE scheme. This results in a more compact representation of both the encrypted database and the set $\mathsf{db}(w)$ compared to the original SSE scheme. Thus, a reduction of the size of the encrypted database and $\mathsf{db}(w)$ also results in a reduction of search time and communication costs.

The Basic Technique: Given $\mathsf{db}(w)$ for any keyword w, we convert $\mathsf{db}(w)$ to a new set c_w, which we call as the *"cover of the keyword"* w. We ensure that, on average, $|c_w|$ is smaller than $|\mathsf{db}(w)|$, while retaining the ability to fully recover $\mathsf{db}(w)$ from c_w. We provide $\widetilde{\mathsf{DB}} = \bigcup_{w \in \mathcal{W}} c_w \times \{w\}$ as an input to any standard secure SSE. Note that $\widetilde{\mathsf{DB}}$ is on average less than the size of $\mathsf{DB} = \bigcup_{w \in \mathcal{W}} \mathsf{db}(w) \times \{w\}$, which is the input to the original SSE. This generic transformation of $\mathsf{db}(w)$ to its cover c_w produces a considerable reduction in both the storage size and result size, resulting in shorter search time and reduced communication size.

The heart of our technique lies in representing $\mathsf{db}(w)$ for each w as a full binary

tree. For each keyword $w \in \mathcal{W}$, we construct a virtual full binary tree where each leaf node is associated with an identifier id_i. We label the leaf nodes with $+$ if the identifier corresponding to it is present in $\mathsf{db}(w)$ and label it with $-$ otherwise. This tree with the labels in the leaf nodes uniquely represents $\mathsf{db}(w)$. We devise a scheme by which this tree can be represented uniquely by a small set of nodes of the tree, and we call this set the cover of the tree.

In this Chapter we define a tree cover and propose several algorithms to generate covers. We do an in-depth combinatorial analysis of these algorithms and prove them to be optimal in our setting. We also provide theoretical estimates of the average size of the covers generated by our algorithm(s) and experimentally validate these theoretical estimates. The cover generation algorithms proposed in this chapter are further used in Chapter 4 to design SSEs. The material in this chapter does not make reference to SSEs and has a complete combinatorial flavor, which can be of independent interest. This chapter was partly published in [31].

3.1 Binary Trees and Tree Covers

3.1.1 Binary Trees

Throughout our thesis, we'll consider a *complete binary tree*, i.e., a binary tree where all levels are present (unless specified otherwise). The *depth* of a node i of a binary tree is the number of edges in the path from i to the root. The root has a depth of 0. In a complete binary tree, there are exactly 2^d nodes at depth d, and consequently, all leaf nodes are at the same depth. The height of a node i is the number of edges in a path from i to the deepest leaf. The height of the tree is the height of the root of the tree. A complete binary tree of height h, has exactly $2^{h+1} - 1$ nodes, with 2^h leaf nodes. The structure of a complete binary tree can be specified only using its height. Let $\mathcal{T}$ be a tree and i a node of $\mathcal{T}$, then by $\mathcal{T}(i)$ we denote the subtree rooted at i. Thus, if i is the root of $\mathcal{T}$, then $\mathcal{T}(i) = \mathcal{T}$ and if i is a leaf node then $\mathcal{T}(i)$ is a tree containing the single node i. Unless specifically mentioned, by a binary tree, we will

mean a complete binary tree.

For convenience, we will denote nodes of a complete binary tree by integers. The root will be denoted by zero. The nodes in depth d will be denoted by 2^d consecutive integers starting with 2^d-1, counting left to right. Thus, for a tree $\mathcal{T}$ of height h, the nodes of $\mathcal{T}$ would be denoted by the set of integers $\mathsf{nodes}(\mathcal{T}) = \{0, 1, \ldots, 2^{h+1} - 2\}$ and the leaf nodes would be denoted by the set $\mathsf{leaves}(\mathcal{T}) = \{2^h - 1, 2^h, \ldots, 2^{h+1} - 2\}$. For any $i \in \mathsf{nodes}(\mathcal{T})$, $\mathsf{leaves}(\mathcal{T}(i)) \subseteq \mathsf{leaves}(\mathcal{T})$ will denote the leaf nodes of the subtree rooted at i.

We will sometimes use an alternative representation of the leave nodes. Given a tree $\mathcal{T}$ of height h with $\mathsf{leaves}(\mathcal{T}) = \{2^h - 1, 2^h, \ldots, 2^{h+1} - 2\}$. Let $\phi_h : \mathsf{leaves}(\mathcal{T}) \to [1, 2^h]$, be a bijection defined as $\phi_h(i) = i - 2^h + 2$. Note that for any $i \in \mathsf{leaves}(\mathcal{T})$, if $j = \phi_h(i)$, then the leaf node i in $\mathcal{T}$ is the j^{th} leaf node from the left. As ϕ is a bijection, we have $\phi_h^{-1} : [1, 2^h] \to \mathsf{leaves}(\mathcal{T})$ defined as $\phi_h^{-1}(j) = j + 2^h - 2$.

3.1.2 Tree Cover

Our main object of interest is a *complete binary tree*, i.e., a binary tree where all levels are present. We fix some basic terminology first.

The *depth* of a node i of a binary tree is the number of edges in the path from i to the root. The root has a depth of 0. In a complete binary tree, there are exactly 2^d nodes at depth d, and consequently, all leaf nodes are at the same depth. The height of a node i is the number of edges in a path from i to the deepest leaf. The height of the tree is the height of the root of the tree. A complete binary tree of height h, has exactly $2^{h+1} - 1$ nodes, with 2^h leaf nodes. The structure of a complete binary tree can be specified only using its height. Let $\mathcal{T}$ be a tree and i a node of $\mathcal{T}$, then by $\mathcal{T}(i)$ we denote the subtree rooted at i. Thus, if i is the root of $\mathcal{T}$, then $\mathcal{T}(i) = \mathcal{T}$ and if i is a leaf node then $\mathcal{T}(i)$ is a tree containing the single node i. Unless specifically mentioned, by a binary tree, we will mean a complete binary tree.

For convenience, we will denote nodes of a complete binary tree by integers. The root will be denoted by zero. The nodes in depth d will be denoted by 2^d consecutive integers starting with 2^d-1, counting left to right. Thus, for a tree $\mathcal{T}$ of height h, the

nodes of $\mathcal{T}$ would be denoted by the set of integers $\mathsf{nodes}(\mathcal{T}) = \{0, 1, \ldots, 2^{h+1} - 2\}$ and the leaf nodes would be denoted by the set $\mathsf{leaves}(\mathcal{T}) = \{2^h - 1, 2^h, \ldots, 2^{h+1} - 2\}$. For any $i \in \mathsf{nodes}(\mathcal{T})$, $\mathsf{leaves}(\mathcal{T}(i)) \subseteq \mathsf{leaves}(\mathcal{T})$ will denote the leaf nodes of the subtree rooted at i.

We will sometimes use an alternative representation of the leave nodes. Given a tree $\mathcal{T}$ of height h with $\mathsf{leaves}(\mathcal{T}) = \{2^h - 1, 2^h, \ldots, 2^{h+1} - 2\}$. Let $\phi_h : \mathsf{leaves}(\mathcal{T}) \to [1, 2^h]$, be a bijection defined as $\phi_h(i) = i - 2^h + 2$. Note that for any $i \in \mathsf{leaves}(\mathcal{T})$, if $j = \phi_h(i)$, then the leaf node i in $\mathcal{T}$ is the j^{th} leaf node from the left. As ϕ_h is a bijection, we have $\phi_h^{-1} : [1, 2^h] \to \mathsf{leaves}(\mathcal{T})$ defined as $\phi_h^{-1}(j) = j + 2^h - 2$.

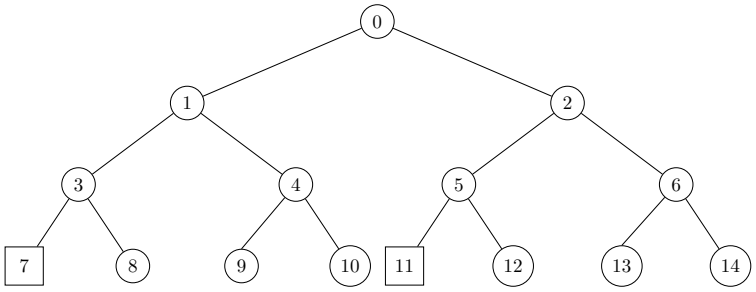


Figure 3-1: The tree $\mathcal{T}$ used in Example 1.

Definition 3.1.1 (Configuration)**.** *Let $\mathcal{T}$ be a complete binary tree of height h and $\mathsf{leaves}(\mathcal{T})$ be the set of leaf nodes of $\mathcal{T}$. A configuration of $\mathcal{T}$ is a function $\Lambda_{\mathcal{T}} : S \to \{+, -\}$, where $S \subseteq \mathsf{leaves}(\mathcal{T})$. The configuration is complete if $S = \mathsf{leaves}(\mathcal{T})$, i.e., a complete configuration of $\mathcal{T}$ is a map $\Lambda_{\mathcal{T}} : \mathsf{leaves}(\mathcal{T}) \to \{+, -\}$. We will represent $\Lambda_{\mathcal{T}}$ by a subset of $\mathsf{leaves}(\mathcal{T}) \times \{+, -\}$.*

Thus, a configuration puts labels from the set $\{+, -\}$ to the leaves of a tree. If all leaves of a tree are labelled then the configuration is called a *complete configuration*. We represent a configuration $\Lambda_{\mathcal{T}} : S \to \{+, -\}$, where $S \subseteq \mathsf{leaves}(\mathcal{T})$ by the set $L = \{(i, s) : i \in S, s = \Lambda_{\mathcal{T}}(i)\}$.

Example 1. *Consider the full binary tree $\mathcal{T}$ of height 3 shown in Figure 3-1. The nodes of $\mathcal{T}$ are elements of the set $\mathsf{nodes}(\mathcal{T}) = \{0, 1, \ldots, 14\}$ and $\mathsf{leaves}(\mathcal{T}) = \{7, 8, \ldots, 14\}$. $L = \{(7, -), (8, +), (9, +), (10, +), (11, -), (12, +), (13, +), (14, +)\}$, is a complete configuration of $\mathcal{T}$, where the nodes 7 and 11 are labeled $-$ and all other*

nodes are labeled $+$*. Whereas, the configuration* $\{(7,-),(8,+),(9,-),(10,+),(11,-)\}$ *is not complete, as the leaf nodes 12, 13 and 14 are not assigned any labels.*

3.1.3 Cover of a Configuration

Our goal is to have a succinct representation of the configuration of a binary tree. For a tree $\mathcal{T}$ the naive representation of its configuration can be done through a subset of $\mathsf{leaves}(\mathcal{T}) \times \{+,-\}$, i.e, by listing all the leaf nodes along with their labels. We are interested in a more compact representation. Informally, a cover is a succinct representation of a configuration.

We define a cover of a configuration through a pair of algorithms *cover generation* and *cover reconstruction*.

Definition 3.1.2 (Syntax of a Tree Cover Scheme)**.** *A tree cover scheme* $\Psi = (\mathsf{CoverGen}, \mathsf{ReConstruct})$ *is a tuple of two algorithms defined as follows.*

- $C \leftarrow \mathsf{CoverGen}(h, L)$*: On input a configuration* L *of a tree* $\mathcal{T}$ *of height* h*,* $\mathsf{CoverGen}$ *outputs a cover* $C \subset \mathsf{nodes}(\mathcal{T}) \times \{+,-\}$*.*
- $L \leftarrow \mathsf{ReConstruct}(h, C)$*: On input the height* h *of a tree* $\mathcal{T}$ *and a cover* $C \subset \mathsf{nodes}(\mathcal{T}) \times \{+,-\}$*,* $\mathsf{ReConstruct}$ *outputs a configuration* $L \subset \mathsf{leaves}(\mathcal{T}) \times \{+,-\}$*.*

Correctness: We say a tree cover scheme Ψ is correct if for any configuration L of a tree of height h,

$$L = \Psi.\mathsf{ReConstruct}(h, \Psi.\mathsf{CoverGen}(h, L)).$$

A tree cover scheme is interesting only if the covers produced by its cover generation algorithm have a size much smaller than the configuration (which it receives as input) for most configurations. The cover generation schemes described later satisfy this property.

We do not yet provide any structural characterisation of a cover of a configuration. A cover is just what is output by a cover generation algorithm, with the guarantee

that there is a corresponding reconstruction algorithm which can reconstruct the configuration from which the cover was generated. This syntactic definition of a cover would be enough for us to develop SSE schemes with desirable properties, but at this point, it may be useful to see what a cover generation algorithm can possibly do.

Let us take the example of Fig 3-1, which represents a complete configuration $L = \{(7,-), (8,+), (9,+), (10,+), (11,-), (12,+), (13,+), (14,+)\}$, of a tree $\mathcal{T}$ of height 3. Nodes drawn in circles that is the nodes $\{8, 9, 10, 12, 13, 14\}$ are marked as $+$ and nodes drawn in squares that is $\{7, 11\}$ are marked as $-$. According to our definition so far, there can be several algorithms for the cover generation which can possibly output different covers for the same tree with the same labeled leaf nodes. We focus on two such simple instances.

1. L itself can be a cover. In this case, on input L the cover generation algorithm simply outputs L as the cover and the reconstruction algorithm on input any set C, just outputs C.

2. The set $C = \{(7,-), (11,-)\}$, can be a cover. Here the cover generation algorithm on input L outputs the set $C = \{(i,-) : (i,-) \in L\}$, i.e., outputs those nodes (along with their labels) which are labeled $-$. The cover reconstruction algorithm on input C labels those nodes in $\mathsf{leaves}(\mathcal{T})$ which are not in C with $+$ and calls those labeled nodes as L_1 and outputs $C \cup L_1$. It is easy to see that this reconstruction always works when L is a complete configuration.

The above two instances are trivial covers. Note that in case (2) we already have a cover whose size is much smaller than the configuration, at least for the example that we consider. In Section 3.2, we systematically study much more complex cover generation algorithms, which, on average, give covers whose sizes are smaller than the configuration. For designing dynamic SSEs a notion of covers for a dynamic configuration would be necessary which we provide in the next subsection. The design of the SSEs presented in Sections 4.1.2 and 4.2 only assumes the definitions of cover generation and reconstruction algorithms. Exact instances of cover generation

algorithms, as discussed in Section 3.2, are not required to follow the material in Sections 4.1.2 and 4.2.

3.1.4 Cover of a Dynamic Configuration

We will be interested in trees where the configuration is dynamic, i.e., we may start with an initial tree along with a configuration, and the tree may change by more nodes getting added to it and/or by the leaves changing labels.

For the sake of modelling dynamic trees, we will use the alternative representation of leaf nodes. Recall for a tree $\mathcal{T}$ of height h, if $i \in \mathsf{leaves}(\mathcal{T}) = [2^h - 1, 2^{h+1} - 2]$, then $\phi_h(i) = i - 2^h + 2$, is the position of the leaf i from the left. Let for a configuration L of tree $\mathcal{T}$ with height h, we define

$$\phi_h(L) = \{(\phi_h(i), s) : (i, s) \in L\}.$$

Thus, $\phi_h(L)$ is just a different representation of the configuration L, where the leaf nodes are specified by their position from the left. Similarly, for any $S \subseteq [1, 2^h] \times \{+, -\}$, we define $\phi_h^{-1}(S) = \{(\phi_h^{-1}(i), s) : (i, s) \in S\}$.

Let T_a, T_b be two complete binary trees of height h_a and h_b respectively. Let L_a and L_b be complete configurations of T_a and T_b respectively. Let $\tilde{L}_a = \phi_{h_a}(L_a)$ and $\tilde{L}_b = \phi_{h_b}(L_b)$. Let $s \in \{+, -\}$. If $s = +$, then $\bar{s} = -$ and vice versa. We define two sets $\tilde{D}$ and $\tilde{A}$ as

$$\tilde{D} = \left\{(i, s) \in \tilde{L}_b : (i, \bar{s}) \in \tilde{L}_a\right\}.$$
$$\tilde{A} = \left\{(i, s) \in \tilde{L}_b : (i, s) \notin \tilde{L}_a \bigwedge (i, \bar{s}) \notin \tilde{L}_a\right\}.$$

Note, the set $\tilde{D}$ contains those labeled leaf nodes in $\tilde{L}_a$ whose labels have changed in $\tilde{L}_b$ and $\tilde{A}$ contains those labeled nodes in $\tilde{L}_b$ which are not present in $\tilde{L}_a$. As, both L_a and L_b are complete configurations of the trees T_a and T_b, then by our definitions

of $\tilde{D}$ and $\tilde{A}$ they are disjoint. We define the *change in configuration* of L_a and L_b as

$$\Delta(L_a, L_b) = \phi_{h_b}^{-1}\left(\tilde{D} \cup \tilde{A}\right). \tag{3.1}$$

Consider a finite sequence of trees $T_1, T_2, \ldots, T_\ell$, where for each i $(1 \leq i \leq \ell - 1)$, and their corresponding complete configurations $L_1, \ldots, L_\ell$. Let

$$L_i^\delta = \Delta(L_i, L_{i+1}). \tag{3.2}$$

Note that, L_i^δ represents a configuration (not necessarily, a complete one) of the tree T_{i+1}. It is easy to see that given T_i along with L_i^δ one can reconstruct L_{i+1}, i.e., the complete configuration of T_{i+1}.

We assume a scenario where we have an initial tree T_1 with a certain complete configuration and over time the tree changes where new labeled nodes are added to the tree or the labels in its leaf node change. We are given the initial tree and the changes that take place in each step, i.e., we have access to T_1 along with $L_1^\delta, L_2^\delta, \ldots, L_{\ell-1}^\delta$.

A dynamic cover generation algorithm outputs a cover on input a configuration. A dynamic cover reconstruction algorithm when given a sequence of covers generates a configuration.

Definition 3.1.3 (Dynamic Tree Cover Scheme)**.** *A dynamic tree cover scheme* $\Psi_d = (\mathsf{dCoverGen}, \mathsf{dReConstruct})$ *is a tuple of two algorithms defined as follows.*

- $C \leftarrow \mathsf{dCoverGen}(h, L)$*: On input the height of the tree* h *and a configuration* L*,* dCoverGen *outputs a cover* C*.*

- $L \leftarrow \mathsf{dReConstruct}\left(\{h_i, C_i\}_{i \in [\ell]}\right)$*: On input a sequence of tree height* h_i *and corresponding cover* C_i*,* dReConstruct *outputs a configuration* L*.*

Correctness: Let $L_1, L_2, \ldots, L_\ell$ be as before, let $\emptyset$ denote the empty configuration corresponding to an empty tree and let

$$\begin{aligned} L_1^\delta &= \Delta(\emptyset, L_1), \\ L_i^\delta &= \Delta(L_i, L_{i+1}), 2 \leq i \leq \ell - 1. \end{aligned}$$

We say, Ψ_d is correct if for all $j \leq \ell$,

$$\Psi_d.\mathsf{dReConstruct}\left(\{h_i, \mathsf{dCoverGen}\left(h_i, L_i^\delta\right)\}_{i\in[j]}\right) = L_j.$$

3.2 Cover Generation Algorithms

For ease of exposition, we will sometimes impose colors on the nodes of the trees. For a node i, $\mathsf{color}(i)$ will denote its color. For a node i, $\mathsf{leftChild}(i)$ and $\mathsf{rightChild}(i)$ will denote its left and right child respectively. Recall, that configuration of a tree $\mathcal{T}$ is a set of labeled leaf nodes of $\mathcal{T}$, thus elements of a configuration are ordered pairs (i, s) where i is a node and $s \in \{+, -\}$ is its label, we will sometimes denote the label of i by $\mathsf{sign}(i)$.

We call a configuration L a *pure configuration* if, for every $(i, s) \in L$, s is the same, i.e., in a pure configuration, every node is either labeled $+$ or $-$. A configuration which is not pure is called a *mixed configuration*.

3.2.1 Pure Cover: A Tree Cover Scheme for Pure Configurations

For the algorithms that follow, we assume that each tree node i is endowed with two fields $\mathsf{color}(i)$ and $\mathsf{sign}(i)$.

We start with a simple scheme which generates covers only for pure configurations. Let L_p be a pure configuration of a tree $\mathcal{T}$, and we want to construct a cover of L_p. As a first step, we color the nodes of the tree according to the scheme described in the algorithm in Figure 3-2. We start from the leaf nodes and color each node which

is in the configuration with the color "GREEN". We proceed with the non-leaf nodes starting from the level just above the leaf nodes and color a node GREEN if both of its children are colored GREEN.

PureColoring(h, L_p):

01. Initialize a tree $\mathcal{T}$ with height h,
    where the nodes of the tree has no color.
02. **for** $i \leftarrow 2^h - 1$ to $2^{h+1} - 2$ $\quad \triangleright$ leaf nodes
03. **if** $(i, s) \in L_p$
04. color$(i) \leftarrow$ GREEN, sign$(i) \leftarrow s$
05. **for** $i \leftarrow 2^h - 1$ to 0 of $\mathcal{T}$ $\quad \triangleright$ non-leaf nodes
06. **if** color(leftChild(i)) = color(rightChild(i)) = GREEN
07. color$(i) \leftarrow$ GREEN, sign$(i) \leftarrow$ sign(leftChild(i))
08. **return** $\mathcal{T}$

Figure 3-2: The coloring scheme for pure configurations

If we apply the algorithm PureColoring(h, L_p) on tree $\mathcal{T}$ with a pure configuration L_p, its nodes either get colored GREEN or they are without color.

Definition 3.2.1 (Top Node)**.** *A colored node in a tree $\mathcal{T}$ is called a* top *node if the path from that node to the root does not contain any other colored node.*

We call the set of top nodes in a tree $\mathcal{T}$ with a pure configuration L_p as the cover of the configuration.

With the above characterization of a cover of a pure configuration, we formulate an algorithm to construct one in Figure 3-3. Initially, the algorithm assigns an empty set X_i to each node i of the tree. For any leaf node j, if the leaf node is colored GREEN, then X_j is set to the singleton set $\{(j, \text{sign}(j))\}$, otherwise X_j remains the empty set. For any non-leaf node i, if the node is colored GREEN, then X_i is set to $\{(i, \text{sign}(i))\}$, else X_i is set as $X_{\text{leftChild}(i)} \bigcup X_{\text{rightChild}(i)}$. Finally, the algorithm returns X_0, i.e., the set associated with the root node.

Proposition 3.2.1. PureCoverGen(h, L_p) *returns the top nodes of a tree of height h with configuration L_p.*

Proof. According to our coloring scheme and the definition of a top node, the following is true for any top node i.

```
PureCoverGen(h, L_p):
01. T ← PureColoring(h, L_p)
02. for i ← 0 to 2^{h+1} − 2
03.     X_i ← ∅
04. for i = 2^h − 1 to 2^{h+1} − 2      ▷ leaf nodes
05.     if color(i) = GREEN
06.         X_i ← {(i, sign(i))}
07. for i ← 2^h − 2 to 0      ▷ every non-leaf node i
08.     if color(i) = GREEN
09.         X_i ← {(i, sign(i))}
10.     else
11.         X_i ← X_{leftChild(i)} ⋃ X_{rightChild(i)}
12. return X_0
```

Figure 3-3: Pure cover generation

1. Both children of i are GREEN. Our coloring scheme guarantees this.

2. The sibling of i is not GREEN, as otherwise, our coloring scheme will make the immediate ancestor of i also GREEN, which violates the condition that i is a top node.

3. No ancestor of i is GREEN, as i is a top node.

Based on the above observations, it follows that if i is a top node, then $X_i = \{(i, s)\}$ (see lines 6 and 9 of the Algorithm in Figure 3-3). Further, for any ancestor j of i, X_j contains $\{(i, s)\}$ (see line 11 of Figure 3-3). As the root (i.e., node 0) is also an ancestor of i hence X_0 contains (i, s). Thus, the set returned by $\mathsf{PureCoverGen}(h, L_p)$ contains all top nodes. Conversely, following the same arguments, it is easy to see that if $(i, s) \in X_0$, then i is a top node. □

We list some additional properties of covers generated by the cover generation algorithm PureCoverGen which are immediate from the algorithm.

Proposition 3.2.2. *Let L_p be a pure configuration of a tree $\mathcal{T}$ of height h, and let $C_p = \mathsf{PureCoverGen}(h, L_p)$ then the following are true.*

1. *If $(i, s) \in C_p$ and i is a leaf node then $(i, s) \in L_p$.*

2. *If $(i, s) \in C_p$ and i is not a leaf node then every leaf node of the subtree rooted at i occurs in L_p with sign s.*

3. *If i and i' be siblings in the tree $\mathcal{T}$ then both (i, s), (i', s) cannot be in C_p.*

PureReConstruct(h, C_p):

01. Initialize a tree $\mathcal{T}$ with height h,
    where the nodes of the tree has no color.
02. Initialize $L_p \leftarrow \emptyset$
03. **for** every node $i = 0$ to $2^h - 2$ $\quad \triangleright$ non-leaf nodes
04. **if** $(i, s) \in C_p$
05. **for** all $j \in \mathsf{leaves}(\mathcal{T}(i))$
06. $L_p \leftarrow L_p \cup \{(j, s)\}$
07. **return** L_p

Figure 3-4: Pure cover reconstruction

Now, given a pure cover C_p, corresponding to a pure configuration L_p, a reconstruction algorithm works as follows (see Figure 3-4). For every non-leaf node i in the cover, the reconstruction algorithm assigns the same sign to all the leaf nodes of the sub-tree rooted at node i and adds those leaf nodes along with their sign to the configuration L_p. And for every leaf node in the cover, it adds the node with its sign to the configuration.

The following proposition, which is easy to verify, asserts that the cover generation scheme is correct.

Proposition 3.2.3. *Let L_p be a pure configuration and $C_p = \mathsf{PureCoverGen}(h, L_p)$ and $L'_p = \mathsf{PureReConstruct}(h, C_p)$, then $L_p = L'_p$.*

3.2.2 Mixed Cover: Generating a Smaller Sized Cover

In the previous section we discussed a scheme to generate covers assuming the configuration to be pure. In this section, we remove this restriction. Let L be a complete configuration for a tree $\mathcal{T}$ of height h. Hence, L contains all leaf nodes of $\mathcal{T}$ along with their sign, i.e.,

$$L = \{(i, \mathsf{sign}(i)) : i \in \mathsf{leaves}(\mathcal{T})\},$$

where $\mathsf{sign}(i) \in \{+, -\}$. Such a complete configuration can be naturally decomposed into pure configurations L_g and L_r where,

$$L_g = \{(i, +) : (i, +) \in L\}, \quad L_r = \{(i, -) : (i, -) \in L\}.$$

As L_g and L_r are pure configurations, we can use our cover generation algorithm for computing the covers of the pure configurations L_g and L_r as $C_g = \mathsf{PureCoverGen}\ (h, L_g)$, $C_r = \mathsf{PureCoverGen}\ (h, L_r)$. Note, that any one of C_g or C_r can be used as a cover of the complete configuration L. The reconstruction would be simple. We would reconstruct the pure configuration L_g (respectively L_r) from C_g (respectively C_r) and assign the opposite sign to all other leaf nodes of the tree. Thus, we can choose the one with smaller number of elements among C_g and C_r as the cover of L.

The above procedure would generate a correct cover for L, but we seek to find a more succinct cover. The above procedure yields a cover which contains nodes of the same sign, and hence we call such a cover as a *pure cover*. The procedure that we are about to describe will contain nodes with both signs and hence we name this algorithm a *mixed cover* algorithm.

```
MixedColoring(h, L_c):
01. Initialize a tree T with height h,
    where the nodes of the tree has no color.
02. for i ← 2^h − 1 to 2^{h+1} − 2
03.   if (i, +) ∈ L_c
04.     color(i) ← GREEN
05.   else
06.     color(i) ← RED
07. for i ← 2^h − 2 to 0
08.   if color(leftChild(i)) = color(rightChild(i)) = GREEN
09.     color(i) ← GREEN
10.   if color(leftChild(i)) = color(rightChild(i)) = RED
11.     color(i) ← RED
12. return T
```

Figure 3-5: Coloring scheme for mixed covers

As before, the heart of the algorithm is a coloring scheme $\mathsf{MixedColoring}(h, L_c)$ which is described in Figure 3-5. The algorithm takes in a complete configuration

(L_c) for a tree $\mathcal{T}$ of height h, and colors the nodes of $\mathcal{T}$ as GREEN or RED. The algorithm examines the nodes level wise starting from the lowest level, i.e., the leaf nodes. All leaf nodes with sign + are colored GREEN and leaf nodes with sign − are colored RED. A non-leaf node gets the color GREEN if both its children are colored GREEN and gets the color RED if both its children are RED. Other nodes remain uncolored.

We discuss the main idea of our cover generation scheme next. It may be helpful to consider an example shown in Figure 3-6 all along. The number of leaf nodes in the tree $n = 16$. Consider that the nodes labeled + are $L_g = \{15, 16, 18, 19, 20, 23\}$, and the nodes labeled − are $L_r = \{17, 21, 22, 24, 25, 26, 27, 28, 29, 30\}$. According to our coloring algorithm of Figure 3-5, the nodes $\{15, 16, 18, 19, 20, 23\}$ will receive the color GREEN (denoted by shaded circles in the Figure 3-6) and the nodes $\{17, 21, 22, 24, 25, 26, 27, 28, 29, 30\}$ will be colored RED (shown by shaded squares in the figure).

Let $\mathcal{T}$ be a tree of height h with a complete configuration L_c and colored using MixedColoring(h, L_c). Recall that a node i is a top node of the colored tree $\mathcal{T}$ if i is colored and there is no colored node in the path from i to the root. In our example in Figure 3-6 the nodes $7, 17, 18, 9, 10, 23, 24, 12$, and 6 are top nodes. Note that, all leaf nodes of the subtree rooted at a top node i have the same color as that of i.

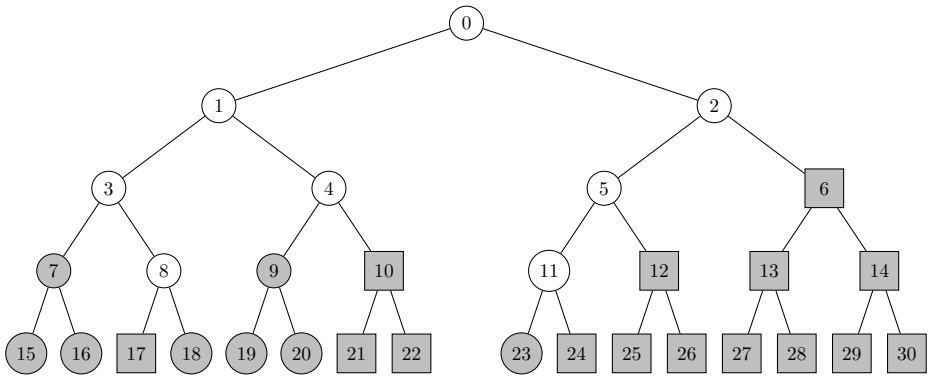


Figure 3-6: An example of mixed cover: The leaf nodes represented by circles are assumed to bear the sign +, the leaf nodes represented by squares bear the sign −. The shaded circles represent nodes colored green and the shaded squares are nodes colored red.

The main intuition of our mixed cover generation algorithm is the following. Let i be a top node for a colored tree $\mathcal{T}$. Hence, the color of i is enough to determine the color of all the leaf nodes of the subtree rooted at i. In fact, in many cases, a top node i along with its color can be used to uniquely color the leaf nodes of a tree much bigger than the subtree rooted at i. For example, in Figure 3-6, node 10 can be used to reconstruct the tree rooted at node 4: we will assign the sign of 10 to all of its leaf nodes and the opposite sign to all the other leaf nodes of the subtree rooted at node 4. We want to make this intuition more concrete. We introduce some definitions for this purpose.

Definition 3.2.2 (Representable Top Nodes)**.** *Let $\mathcal{T}$ be a tree of height h with complete configuration L_c and colored by* $\mathsf{MixedColoring}(h, L_c)$*. A set of top nodes S of $\mathcal{T}$ is called representable, if the following holds*

1. *All nodes in S are of same color $c \in \{\text{GREEN,RED}\}$.*

2. *There exists a subtree $\mathcal{T}'$ (of $\mathcal{T}$) containing all nodes in S, such that all leaf nodes of $\mathcal{T}'$ other than the leaf nodes of the trees rooted at nodes in S have a color different from c.*

If S is representable then any $\mathcal{T}'$ satisfying the property (2) above is said to represent S.

Definition 3.2.3 (Representative Tree)**.** *Let $\mathcal{T}$ be a tree of height h with complete configuration L_c and colored by* $\mathsf{MixedColoring}(h, L_c)$*. S be a representable set of top nodes of $\mathcal{T}$. The largest subtree of $\mathcal{T}$ which represents S is called the representative tree of S. By $\mathsf{rep}_{\mathcal{T}}(S)$ we will denote the root of the representative tree of S. If S is a singleton set containing only i, we will denote the root of their representative tree of S by $\mathsf{rep}_{\mathcal{T}}(i)$.*

From the definition, it is immediate that every representable set of top nodes will have a representative tree, moreover every singleton set containing a top node is representable and thus has a representative tree.

In our example tree $\mathcal{T}$ in Figure 3-6 we have the set $\{17, 10\}$ is representable and $\mathsf{rep}_{\mathcal{T}}(\{17, 10\}) = 1$, whereas $\{17, 24\}$ is not representable. Moreover, we have $\mathsf{rep}_{\mathcal{T}}(7) = 7$, $\mathsf{rep}_{\mathcal{T}}(17) = 3$, $\mathsf{rep}_{\mathcal{T}}(9) = \mathsf{rep}_{\mathcal{T}}(10) = 4$, $\mathsf{rep}_{\mathcal{T}}(23) = 2$, $\mathsf{rep}_{\mathcal{T}}(12) = 12$ and $\mathsf{rep}_{\mathcal{T}}(6) = 6$.

Definition 3.2.4 (Independent Nodes)**.** *Let S_1, S_2 be two sets of representable top nodes of a colored tree $\mathcal{T}$ and let $\mathsf{rep}_{\mathcal{T}}(S_1) = i_1$ and $\mathsf{rep}_{\mathcal{T}}(S_2) = i_2$. We say S_1, S_2 are independent if $\mathcal{T}(i_1)$ and $\mathcal{T}(i_2)$ are disjoint.*

In our example, the sets {7},{17} are not independent whereas {17}, {10} and {23}, {17} are independent.

Definition 3.2.5 (Span)**.** *Let S be a set of top nodes of a tree $\mathcal{T}$ colored by* MixedColoring *(h, L). We say that S spans $\mathcal{T}$, if S can be decomposed into $S = S_1 \cup S_2 \cup \cdots \cup S_k$, for some $k \leq |S|$ such that the following holds*

1. *For all $i, j \in [k]$, $S_i \cap S_j = \emptyset$.*
2. *For all $i \in [k]$, S_i is representable.*
3. *For all $i, j \in [k]$, $i \neq j$, S_i and S_j are independent.*
4. *The union of the leaf nodes of the representative trees of the sets $S_1, \ldots, S_k$, gives the leaves of the tree $\mathcal{T}$, i.e.,*

$$\bigcup_{i \in [k]} \mathsf{leaves}\left(\mathcal{T}\left(\mathsf{rep}_{\mathcal{T}}(S_i)\right)\right) = \mathsf{leaves}(\mathcal{T}).$$

Observe that if X spans a colored tree $\mathcal{T}$, then the nodes in X along with their colors contain enough information to reconstruct the complete configuration of $\mathcal{T}$. Thus, for a given colored tree our goal is to find a set X of top nodes of minimum size which spans the tree. For the reconstruction, it is necessary to just decompose the span into representable sets and determine the representative tree for each such representable set.

Consider any node i of a colored tree $\mathcal{T}$. If i is colored and is of color $c \in \{\text{RED},\text{GREEN}\}$, then i along with its color/sign spans the tree $\mathcal{T}(i)$. If i is un-colored, then the following statements hold and are easy to verify.

Proposition 3.2.4. *Let i be any un-colored node in a colored tree $\mathcal{T}$ and ρ and λ be the right child and left child of i, respectively. Let X_ρ and X_λ span $\mathcal{T}(\rho)$ and $\mathcal{T}(\lambda)$ respectively. Then either both X_ρ and X_λ individually spans $\mathcal{T}(i)$ or $X_\lambda \cup X_\rho$ spans $\mathcal{T}(i)$.*

Proposition 3.2.5. *Let i be any un-colored node in a colored tree $\mathcal{T}$ and ρ and λ be the right child and left child of i, respectively. Let X spans $\mathcal{T}(i)$ and let $X = X_\lambda \cup X_\rho$, where $X_\lambda \subseteq \mathsf{nodes}(\mathcal{T}(\lambda))$ and $X_\rho \subseteq \mathsf{nodes}(\mathcal{T}(\rho))$. For $j \in \{\lambda, \rho\}$, if $X_j \neq \emptyset$, then X_j spans $\mathcal{T}(j)$. Moreover, if $X_\lambda = \emptyset$ then X_ρ contains nodes of the same color and all leaves of $\mathcal{T}(\lambda)$ are of color different from that of the nodes of X_ρ.*

The above observation is central to our algorithm $\mathsf{MixedCoverGen}(h, L_c)$ for generating covers described in Figure 3-7. The algorithm finds a cover for a tree $\mathcal{T}$ of height h with a complete configuration L_c. The algorithm assigns to each node i, three sets namely $X_{i,g}$, $X_{i,r}$, $X_{i,m}$. $X_{i,g}$ and $X_{i,r}$ contains the GREEN and RED top nodes of the subtree rooted at i. $X_{i,m}$ is empty if i is a leaf node, otherwise we set

$$Y \leftarrow \mathsf{MixedUnion}(i).$$

The procedure $\mathsf{MixedUnion}(i)$, when i is not a leaf node, is described in the right column of Figure 3-7. $\mathsf{MixedUnion}(i)$ computes the following collection of sets

$$\mathcal{C} = \Big\{ X_{\mathsf{leftChild}(i),x} \bigcup X_{\mathsf{rightChild}(i),y} : x, y \in \{r, g, m\}; \\ x \neq y; X_{\mathsf{leftChild}(i),x}, X_{\mathsf{rightChild}(i),y} \neq \emptyset \Big\},$$

and returns the set Y of minimum cardinality from $\mathcal{C}$. If $|Y|$ is smaller than both $|X_{i,g}|$ and $|X_{i,r}|$, then $X_{i,m}$ is set to Y otherwise it remains empty. Finally, the algorithm outputs $C = \mathsf{minCard}(X_{0,r}, X_{0,g}, X_{0,m})$. The function $\mathsf{minCard}(X_1, \ldots, X_n)$, for finite

sets $X_1, X_2, \ldots, X_n$ gives a nonempty set of minimum cardinality among the sets $X_1, X_2, \ldots, X_n$.

Theorem 3.2.1. *Let $\mathcal{T}$ be a tree of height h, and L_c be a complete configuration of $\mathcal{T}$. Let* $\mathsf{MixedCoverGen}(h, L_c)$ *compute the sets $X_{i,g}, X_{i,r}, X_{i,m}$ for each node i of $\mathcal{T}$ as described in Figure 3-7. Let $C_i = \{X_{i,g}, X_{i,r}, X_{i,m}\}$. Then the following are true.*

1. *C_i contains at least one non-empty set.*

2. *Any non-empty set $X \in C_i$ spans $\mathcal{T}(i)$.*

3. $\mathsf{minCard}(C_i)$ *is the smallest set of nodes which spans $\mathcal{T}(i)$.*

Proof. The proof of (1) is immediate, as at least one of $X_{i,r}$ or $X_{i,g}$ must be non-empty as L_c is a complete configuration of $\mathcal{T}$ and hence all the leaves of $\mathcal{T}$ are colored.

(2) Directly follows from the description of the algorithm and Proposition 3.2.4.

We prove (3) by induction on the height of a node. Let i be of height 0, i.e., a leaf node and without loss of generality let i be colored GREEN, thus $X_{i,g} = \{(i, +)\}$ and $X_{i,r} = X_{i,m} = \emptyset$, and $X_{i,g}$ is the smallest set which spans $\mathcal{T}(i)$ as $|X_{i,g}| = 1$. This serves as the base case. As induction hypothesis consider that (3) is true for all nodes at height less than ℓ. Consider, a node i at height ℓ. As $\ell > 0$, i have two children say λ and ρ. By our induction hypothesis, $X_\lambda = \mathsf{minCard}(X_{\lambda,r}, X_{\lambda,g}, X_{\lambda,m})$ and $X_\rho = \mathsf{minCard}(X_{\rho,r}, X_{\rho,g}, X_{\rho,m})$ are the smallest sets which spans $\mathcal{T}(\lambda)$ and $\mathcal{T}(\rho)$ respectively. Now we have a few cases to consider:

> **Case 1**: i is colored. Without loss of generality let $\mathsf{color}(i) =$ GREEN, then $X_{i,g} = \{(i, +)\}$ and $X_{i,r}$ and $X_{i,m}$ are empty. Moreover, as $X_{i,g} = \{(i, +)\}$ contains a single node which is the root of $\mathcal{T}(i)$, thus $X_{i,g}$ is the smallest set which spans $\mathcal{T}(i)$.
>
> **Case 2**: i is not colored. Let $X = \mathsf{minCard}(C_i)$, and for the sake of contradiction let Y span $\mathcal{T}(i)$ and let $|Y| < |X|$. Let $Y = Y_\lambda \cup Y_\rho$ where Y_j contains nodes in $\mathcal{T}(j)$ for $j \in \{\lambda, \rho\}$. We consider two subcases:

(a) Both Y_λ and Y_ρ are non-empty. By Proposition 3.2.5, Y_λ and Y_ρ spans $\mathcal{T}(\lambda)$ and $\mathcal{T}(\rho)$ respectively. Now, we claim that either $|Y_\lambda| < |X_\lambda|$ or $|Y_\rho| < |X_\rho|$, and this contradicts our induction hypothesis. Finally, to see why our claim is correct, notice that by our algorithm $X_\lambda \cup X_\rho \in C_i$, and as $|Y| < |X|$ and $X = \mathsf{minCard}(C_i)$ it cannot be that $|Y_\lambda| \geq |X_\lambda|$ and $|Y_\rho| \geq |X_\rho|$.

(b) One of Y_λ and Y_ρ is empty. Without loss of generality, let $Y_\rho = \emptyset$, i.e., $Y = Y_\lambda$. Then by Proposition 3.2.5, Y_λ must contain nodes of the same color, say GREEN, and all leaves of $\mathcal{T}(\rho)$ must be RED. Hence, node ρ must be RED and all nodes in $\mathsf{leaves}(\mathcal{T}(\lambda)) \setminus Y_\lambda$ must also be RED. In this configuration, the set $X_{i,g}$ computed by MixedCoverGen, will contain the same nodes as in Y_λ. Hence, we have $Y = Y_\lambda \in C_i$, which contradicts $|Y| < |X|$, as $X = \mathsf{minCard}(C_i)$.

□

MixedCoverGen(h, L_c):	MixedUnion(i):
01. $\mathcal{T} \leftarrow$ MixedColoring(h, L_c)	01. cnt $\leftarrow 0$
02. **for** every node i of tree $\mathcal{T}$	02. **for** $x \in \{r, g, m\}$
03. $X_{i,g}, X_{i,r}, X_{i,m} \leftarrow \emptyset$	03. **for** $y \in \{r, g, m\}$
04. **for** $i \leftarrow 2^h - 1$ to 0	04. $\lambda \leftarrow$ leftChild(i)
05. **if** color(i) = GREEN	05. $\rho \leftarrow$ rightChild(i)
06. $X_{i,g} \leftarrow X_{i,g} \bigcup \{(i, +)\}$	06. **if** $(x \neq y) \bigwedge X_{\lambda,x} \neq \emptyset \bigwedge X_{\rho,y} \neq \emptyset$
07. **else-if** color(i) = RED	07. $Y_{\mathsf{cnt}} \leftarrow X_{\lambda,x} \bigcup X_{\rho,y}$
08. $X_{i,r} \leftarrow X_{i,r} \bigcup \{(i, -)\}$	08. cnt $\leftarrow$ cnt $+ 1$
09. **else**	09. $X \leftarrow \mathsf{minCard}(Y_0, Y_1, \ldots, Y_{\mathsf{cnt}})$
10. $X_{i,g} \leftarrow X_{\mathsf{leftChild}(i),g} \bigcup X_{\mathsf{rightChild}(i),g}$	10. **return** X
11. $X_{i,r} \leftarrow X_{\mathsf{leftChild}(i),r} \bigcup X_{\mathsf{rightChild}(i),r}$	
12. $Y \leftarrow$ MixedUnion(i)	
13. **if** $\lvert Y\rvert < \min\{\lvert X_{i,g}\rvert, \lvert X_{i,r}\rvert\}$	
14. $X_{i,m} \leftarrow Y$	
15. **return** minCard$(X_{0,g}, X_{0,r}, X_{0,m})$	

Figure 3-7: Mixed cover generation algorithm

Theorem 3.2.1 asserts that MixedCoverGen(h, L_c) outputs the smallest possible set X which spans a tree of height h with a complete configuration L_c. Now our goal is

to regenerate the configuration of a tree of height h, given h and a set C which spans the tree and was produced by $\mathsf{MixedCoverGen}(h, L_c)$. The reconstruction algorithm is presented in Figure 3-8.

First, note that it may be the case that the algorithm $\mathsf{MixedCoverGen}(h, L_c)$ generates a pure cover, i.e., it generates C where all nodes in C are labeled with the same sign. In such a case either $C = X_{0,r}$ or $C = X_{0,g}$, i.e., C contains all top nodes of a single color. In this case, the reconstruction is simple, as the representative tree for C is the complete tree, i.e., $\mathsf{rep}_{\mathcal{T}}(C) = 0$. For reconstruction, the following procedure would suffice: For every $(i, \mathsf{sign}(i)) \in C$, the leaves of the sub-tree $\mathcal{T}(i)$ are labeled with $\mathsf{sign}(i)$ and the rest of the leaf nodes are labeled with the opposite sign. These steps are done in lines 5 to 8 of the algorithm described in Figure 3-8.

If C is a mixed cover, i.e., C contains nodes of both signs then $C = X_{0,m}$. In this case, the reconstruction procedure is a bit more involved. For convenience, we introduce some additional notations. For any node i in a tree $\mathcal{T}$, let $\mathsf{path}(i)$ denote the sequence of nodes in the unique path from i to the root of $\mathcal{T}$. For any node $j \neq i$ of $\mathsf{path}(i)$, $\mathsf{prev}_i(j)$ denotes the node preceding j in the sequence $\mathsf{path}(i)$.

Reconstructing the configuration, given a mixed cover C boils down to decomposing C into disjoint subsets such that each subset is representable. Once such representable subsets are obtained the leaves of the corresponding representative trees can be assigned colors (signs) and this assignment of signs would yield the desired complete configuration.

Let C be the output of $\mathsf{MixedCoverGen}$ (h, L_c) and let C be a mixed cover. Let $(j, s) \in C$, where $s \in \{+, -\}$ and let $\bar{s}$ be the sign opposite to s. Let, ℓ_j be the first node in $\mathsf{path}(j)$ which intersects with $\mathsf{path}(i)$ for some node i such that $(i, \bar{s}) \in C$. As C is a mixed cover, hence C contains at least two elements with different signs and thus for every $(j, s) \in C$, ℓ_j is well-defined.

Let λ and ρ be the left and right children of ℓ_j. Then, if i belongs to $\mathcal{T}(\lambda)$ then j belongs to $\mathcal{T}(\rho)$ and vice versa. Without loss of generality, let i belong to $\mathcal{T}(\lambda)$ also

let J be those nodes in $\mathcal{T}(\rho)$ which belongs to C, i.e.,

$$J = \{(k, \mathsf{sign}(k)) : (k, \mathsf{sign}(k)) \in C, k \in \mathcal{T}(\rho)\}. \tag{3.3}$$

It is easy to see that all nodes in J bears the same sign as that of j, as otherwise ℓ_j cannot be the first node in $\mathsf{path}(j)$ which intersects with the path of another node in C with a sign different from the sign of j. We observe the following

Proposition 3.2.6. *The set J as defined in Eq. (3.3) is representable and $\rho = \mathsf{rep}_{\mathcal{T}}(J)$.*

Proof. As all nodes in J are top nodes and bear the same sign, then in the corresponding colored tree $\mathcal{T}$ all nodes in J have the same color, say GREEN. With reference to the algorithm $\mathsf{MixedCoverGen}((h, L_c))$, it is immediate that $J \subseteq X_{\rho,g}$. We claim that $J = X_{\rho,g}$, i.e., J contains all GREEN top nodes in $\mathcal{T}(\rho)$. As C is a cover of $\mathcal{T}$ and $J \subseteq C$, by Theorem 3.2.1, C spans $\mathcal{T}$, and the nodes in J are the only GREEN top nodes in $\mathcal{T}(\rho)$ which are in C. If there are GREEN top nodes in $\mathcal{T}(\rho)$ which are not in J then C cannot span $\mathcal{T}$. Thus, J is representable and $\mathcal{T}(\rho)$ represents J.

We are left to show that $\mathcal{T}(\rho)$ is the largest tree that represents J, i.e., $\mathsf{rep}_{\mathcal{T}}(J) = \rho$. The smallest tree larger than $\mathcal{T}(\rho)$ which contains $\mathcal{T}(\rho)$ is $\mathcal{T}(\ell_j)$. We will argue that $\ell_j \neq \mathsf{rep}_{\mathcal{T}}(J)$. Note that, $i \in C$ and i belongs to the left subtree $\mathcal{T}(\lambda)$ of $\mathcal{T}(\ell_j)$ and i has color different from j, i.e., i is colored RED. If $\ell_j = \mathsf{rep}_{\mathcal{T}}(J)$, then a set containing J and another set containing i cannot be independent, which implies that both nodes in J and the node i cannot be in C which by Theorem 3.2.1 spans $\mathcal{T}$. □

The above proposition is central to the reconstruction algorithm presented in Figure 3-8 when C is a mixed cover. Lines 10-14 of the algorithm presented in Figure 3-8 finds the node ℓ_j (as described above) for each j in C and thus decomposes C into representative sets J as asserted in Proposition 3.2.6. This process is iterated until all leaves of the tree are labeled.

MixedReConstruct(h, C):

01. Initialize a tree $\mathcal{T}$ with height h, where the nodes of the tree has no color.
02. $L \leftarrow \emptyset$
03. **for** all $(i, s) \in C$
04. **for** all $k \in \mathsf{leaves}(\mathcal{T}(i))$
05. $L \leftarrow L \cup \{(k, s)\}$
05. **if** C is a pure cover with sign $s \in \{+, -\}$
06. **for** all $k' \in \mathsf{leaves}(\mathcal{T})$
07. **if** $(k', s) \notin L$
08. $L \leftarrow L \cup \{(k', \bar{s})\}$
09. **else**
10. **for** each $(i, s) \in C$
11. find the first ancestor node j of i in $\mathsf{path}(i)$ such that $j \in \mathsf{path}(l)$ for some $(l, \bar{s}) \in C$
12. **for** all the leaf nodes k of the sub-tree rooted at the node $\mathsf{prev}_i(j)$
13. **if** $(k, s) \notin L$
14. $L \leftarrow L \cup \{(k, \bar{s})\}$
15. **return** L

Figure 3-8: Mixed cover reconstruction

3.2.3 Complexity of Proposed Algorithms

In the MixedCoverGen algorithm (Figure 3-7), the loop in line 4 iterates $N = 2^{h+1} - 1$ times, i.e., for all nodes in the tree. For each iteration, the algorithm computes a minimum-sized cover of the node using the MixedUnion algorithm. This algorithm operates on a single node and determines the minimum-sized mixed cover by taking the union of all possible covers from the left and right children of the target node. Thus, the overall complexity of the mixed cover generation algorithm is determined by the complexity of the MixedUnion algorithm.

The MixedUnion algorithm has a constant-time computational complexity and requires linear space. To achieve this, the algorithm maintains the RED, GREEN, and MIXED covers of each node using linked lists. Each linked list is structured such that its head contains the length of the list and an additional pointer to the tail. This setup allows us to efficiently compute the cover with the minimum cardinality using the length information stored at the head, and a union operation can be performed by manipulating the head and tail pointers of the linked lists being merged. Thus, the MixedUnion operations have constant time complexity. In addition, the size of the linked lists is linear with respect to the number of tree nodes. The algorithm processes

the tree level-wise, and at each level, the total size of all linked lists is linear in the number of tree nodes. Covers for all nodes below the current level can be discarded. Therefore, the time and space complexity of the proposed algorithm MixedCoverGen is $\mathcal{O}(N)$.

The MixedReConstruct algorithm of Figure 3-8 runs for at most the number of elements in the input cover C. The size of the input cover is at most half the number of leaf nodes, i.e., $n/2 = 2^{h-1}$. For each node in the cover, the algorithm examines the path from that node to the root. It searches for the first intersection of this path with another path from a different node in the cover with opposite color. This first intersection determines the representative tree for the target node (the details are described in the algorithm MixedReConstruct of Figure 3-8).

This process can be optimized by pre-computing all paths from the elements in the cover, which requires at most $\mathcal{O}(h)$ time per element. As a result, the overall time complexity of the reconstruction algorithm is $\mathcal{O}(C \cdot h)$. Lastly, the reconstruction algorithm has a space complexity $\mathcal{O}(N)$, since the entire process can be performed directly on the tree structure.

3.2.4 Dynamic Tree Cover Scheme

Recall the setting of a dynamic tree cover scheme as discussed in Section 3.1.4. We have a sequence of trees $T_1, T_2, \ldots, T_\ell$, with their corresponding complete configurations $L_1, L_2, \ldots, L_\ell$, and for $1 \leq i \leq \ell - 1$, L_i^δ is defined as in Equations (3.1) and (3.2). Note each L_i^δ represents a configuration, not necessarily a complete one, of a tree of height h_i, where h_i is the height of the shortest complete binary tree which contains all nodes in L_i^δ. The dynamic cover generation algorithm takes L_i^δ and generates a cover C_i. The property which is required is that, knowing the complete configuration of T_1 and the sequence of covers $C_2, C_3, \ldots, C_\ell$ corresponding to the configurations $L_2^\delta, L_3^\delta, \ldots, L_{\ell-1}^\delta$, the cover reconstruction algorithm can generate the complete configuration of T_n.

The dynamic cover generation algorithm just takes a configuration and produces its cover. If the input configuration L is a complete configuration then the mixed

cover generation algorithm is used to generate the cover. Otherwise, L is decomposed into two sets L_g and L_r, where L_g contains the nodes labeled $+$ and L_r contains the nodes labeled $-$, and the pure cover generation algorithm is used to generate covers C_g and C_r for the configurations L_g and L_r respectively. Finally, $C = C_g \cup C_r$ is produced as the output. The details are shown in Figure 3-9.

```
dCoverGen(h, L):
01. Initialize an empty tree 𝒯 of height h
02. if |L| = 2^h
03.    C ← MixedCoverGen(h, L)
04. else
05.    C, L_g, L_r ← ∅
06.    for all (i, +) ∈ L
07.       L_g ← L_g ∪ {(i, +)}
08.    C_g ← PureCoverGen(h, L_g)
09.    for all (i, −) ∈ L
10.       L_r ← L_r ∪ {(i, −)}
11.    C_r ← PureCoverGen(h, L_r)
12.    C ← C_g ∪ C_r
13. return C
```

Figure 3-9: Dynamic cover generation

To explain the reconstruction procedure we introduce a new operation on configurations. Let L_α and L_β be two configurations for trees with heights h_α and h_β. Let $h_{\max} = \max\{h_\alpha, h_\beta\}$, $\tilde{L_\alpha} = \phi_{h_\alpha}(L_\alpha)$ and $\tilde{L_\beta} = \phi_{h_\beta}(L_\beta)$. We define a new configuration for a tree of height $h_{\max}$ as

$$L_\alpha \rhd_{(h_\alpha, h_\beta)} L_\beta = \phi^{-1}_{h_{\max}}(X \cup Y \cup Z), \tag{3.4}$$

where X, Y, Z are defined as follows:

$$\begin{aligned} X &= \{(i,s) \in \tilde{L}_\alpha : (i,s) \notin \tilde{L}_\beta \text{ and } (i,\bar{s}) \notin \tilde{L}_\beta\} \\ Y &= \{(i,s) \in \tilde{L}_\beta : (i,s) \notin \tilde{L}_\alpha, (i,\bar{s}) \notin \tilde{L}_\alpha\} \\ Z &= \{(i,s) \in \tilde{L}_\beta : (i,\bar{s}) \in \tilde{L}_\alpha\}. \end{aligned}$$

$L_\alpha \rhd_{(h_\alpha, h_\beta)} L_\beta$ is essentially the configuration obtained by overwriting L_α by L_β.

Notice that the set X contains those labeled nodes in L_α which are not present in L_β. Similarly, Y contains those nodes in L_β which are not present in L_α and Z contains those nodes in L_β which are present in L_α but with the opposite label.

For reconstruction, the sequence of covers, along with the corresponding tree heights, is used. Suppose C_1 be the cover of L_1 and for $2 \leq i \leq \ell - 1$, C_i be the cover of L_i^δ. The cover reconstruction algorithm first reconstructs the configuration L_i for each cover C_i and then outputs the configuration

$$L = ((((L_2 \rhd L_2) \rhd L_3) \rhd \cdots) \rhd L_{\ell-1}) .$$

The details are in Figure 3-10.

dCoverReConstruct($\{(h_i, C_i)\}_{i\in[0,\ell-1]}$):

01. $L_0 \leftarrow$ MixedReConstruct(h_0, C_0)
02. **for** $i = 1$ to $\ell - 1$
03. Initialize $X_i, C_p, C_r \leftarrow \emptyset$
04. **for** all $(j, +) \in C_i$
05. $C_p \leftarrow C_p \cup \{(j, +)\}$
06. $L_p \leftarrow$ PureReConstruct(h_i, C_p)
07. **for** all $(j, -) \in C_i$
08. $C_r \leftarrow C_r \cup \{(j, -)\}$
09. $L_r \leftarrow$ PureReConstruct(h_i, C_r)
10. $X_i \leftarrow L_p \cup L_r$
11. $h \leftarrow h_0$, $L \leftarrow L_0$
12. **for** $i = 1$ to $\ell - 1$,
13. $L \leftarrow L \rhd_{(h,h_i)} X_i$
14. $h \leftarrow \max\{h, h_i\}$
15. **return** L

Figure 3-10: Dynamic cover reconstruction

3.3 Expected Cover Size of a Pure Cover

In this section, we provide an analysis of the expected size of a cover returned by the Algorithm in Figure 3-3. Let $\mathcal{T}$ be a full binary tree of height h and thus $\mathcal{T}$ has a total of $n = 2^h$ leaf nodes. Let L_p be a pure configuration of $\mathcal{T}$ such that $|L_p| = r$, and without loss of generality, we assume that all nodes in L_p bear the sign $+$. There

are $\binom{n}{r}$ such configurations possible, and we are interested in the average (expected value of) cover size over all these $\binom{n}{r}$ configurations.

Consider a sequence of $2n-1$ binary random variables $P_0, P_1, \ldots, P_{2n-2}$ corresponding to each node i of $\mathcal{T}$. Define

$$P_i = \begin{cases} 1; & \text{if } \mathsf{color}(i) = \textsc{green} \\ 0; & \text{otherwise.} \end{cases}$$

That is, according to our coloring scheme, $P_i = 1$ denotes if the node i is colored GREEN or not. Consider any node i at the ℓ^{th} level of the tree $\mathcal{T}$, i.e., $i \in \left\{2^\ell - 1, \ldots, 2^{\ell+1} - 2\right\}$. Then, the subtree rooted at node i contains $2^{h-\ell}$ many leaf nodes. The event "$\{P_i = 1\}$" can then be viewed as choosing $2^{h-\ell}$ many BLACK balls from a bag containing a total of n balls, where r many BLACK balls and remaining $n-r$ are WHITE balls. Therefore,

$$\Pr[P_i = 1] = \begin{cases} \frac{\binom{r}{2^{h-\ell}}}{\binom{n}{2^{h-\ell}}} = \eta_{2^{h-\ell}}(n, r); & \text{if } 2^{h-\ell} \leq r \\ 0; & \text{otherwise,} \end{cases}$$

where $\eta_\rho(\nu, \xi) = \frac{\binom{\xi}{\rho}}{\binom{\nu}{\rho}}$ denotes the probability of choosing ρ many BLACK balls from a bag containing ξ many are BLACK balls and the $\nu - \xi$ many WHITE balls. In order to avoid writing the boundary conditions every time, we extend the definition of $\eta_\rho(\nu, \xi)$ in the following way.

$$\eta_\rho(\nu, \xi) \triangleq \begin{cases} \frac{\binom{\xi}{\rho}}{\binom{\nu}{\rho}}; & \text{if } \nu, \xi, \rho \geq 0 \text{ and } \rho \leq \xi \\ 0; & \text{otherwise.} \end{cases}$$

Define another sequence of $2n-1$ binary random variables $X_0, X_1, \ldots, X_{2n-2}$ in the

following manner.

$$X_i \;=\; \begin{cases} 1; & \text{if} \begin{cases} i = \text{ odd and } P_i = 1 \text{ and } P_{i+1} = 0 \\ i = \text{ even and } P_i = 1 \text{ and } P_{i-1} = 0 \end{cases} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the random variable X_i corresponding to the node i contributes 1 to the size of the cover if i is a GREEN node, but its sibling node is not a GREEN node. Assume that i is odd and $2^\ell < i < 2^{\ell+1}$. Then,

$$\begin{aligned} \Pr[X_i = 1] &= \Pr[\{P_i = 1\} \cap \{P_{i+1} = 0\}] \\ &= \Pr[P_{i+1} = 0 | P_i = 1] \cdot \Pr[P_i = 1] \\ &= (1 - \Pr[P_{i+1} = 1 | P_i = 1]) \cdot \Pr[P_i = 1] \\ &= \left(1 - \frac{\binom{r-2^{h-\ell}}{2^{h-\ell}}}{\binom{n-2^{h-\ell}}{2^{h-\ell}}}\right) \cdot \eta_{2^{h-\ell}}(n, r) \\ &= \left(1 - \eta_{2^{h-\ell}}\left(n - 2^{h-\ell}, r - 2^{h-\ell}\right)\right) \cdot \eta_{2^{h-\ell}}(n, r). \end{aligned}$$

Let the random variable X denote the cover size. Then, X can be expressed as

$$X \;=\; X_0 + X_1 + \cdots + X_{2^{h+1}-2}.$$

Assume that $2^\alpha \leq r < 2^{\alpha+1}$. This implies that $P_i = 0$ for all $0 \leq i \leq 2^{h-\alpha} - 2$ with the convention that if $2^{h-\alpha} - 2 < 0$ then such an i does not exist. Then the cover size X is given by,

$$\begin{aligned} X &= X_{2^{h-\alpha}-1} + X_{2^{h-\alpha}} + \cdots + X_{2^{h+1}-2} \\ &= \sum_{i=h-\alpha}^{h} \left(X_{2^i-1} + X_{2^i} + \cdots + X_{2^{i+1}-2}\right). \end{aligned}$$

Therefore, by linearity of expectation, the expected cover size is given by

$$\begin{aligned} E[X] &= \sum_{i=h-\alpha}^{h} \left(E[X_{2^i-1}] + E[X_{2^i}] + \cdots + E[X_{2^{i+1}-2}]\right) \\ &= \sum_{i=h-\alpha}^{h} \left\{2^i \cdot \left(1 - \eta_{2^{h-i}}\left(n - 2^{h-i}, r - 2^{h-i}\right)\right) \cdot \right. \\ &\quad \left. \eta_{2^{h-i}}\left(n, r\right)\right\}, \end{aligned} \tag{3.5}$$

where r denote the size of the pure configuration such that $2^\alpha \leq r < 2^{\alpha+1}$.

3.4 Experimental Results

In this section, we test the performance of our proposed algorithms experimentally. Consider a tree with n leaf nodes, where $n = 2^k$ for some k. Let r leaves of the tree would be labeled $+$ and $n-r$ leaves would be labeled $-$. We are interested in finding the cover size of such a tree. Note that if we fix a tree of size n and r of the leave are $+$, then the tree has $\binom{n}{r}$ different configurations, and each configuration will give rise to a cover of a different size. In our first experiment, we keep n fixed to 32, and for each $0 \leq r \leq 32$, we generate $\binom{n}{r}$ configurations and compute the cover of all these configurations. For each configuration, we generate three types of covers:

1. Pure cover with the nodes labeled $+$, by using the algorithm $\mathsf{PureCoverGen}(\cdot,\cdot)$ described in Figure 3-3.

2. Pure cover with the nodes labeled $-$, also using algorithm $\mathsf{PureCoverGen}(\cdot,\cdot)$ of Figure 3-3.

3. Mixed cover using Algorithm $\mathsf{MixedCoverGen}(\cdot,\cdot)$ described in Figure 3-7.

For each r, we compute the average cover size over all the configurations. These results are summarized in Figure 3-11. The X-axis of the graph shown in Figure 3-11 represents the value r, i.e., the number of identifiers which are present corresponding to the keyword w. The Y-axis represents the average cover size of all $\binom{n}{r}$ configurations corresponding to r. The size of the three types of covers is represented by three

different types of lines. In addition, we have also plotted the line $Y = X$ for the sake of reference.

The following can be immediately observed from the results demonstrated in Figure 3-11:

1. For all three types of covers, the average size of the covers increases with r, and then it decreases.

2. For all values of r, the size of the mixed cover is smaller than the size of the pure covers. The difference is significant when the number of identifiers present is around half of the total number of identifiers.

3. The curves representing the average size of pure cover with nodes labeled + and the mixed cover always lies below the line $Y = X$

We have theoretically calculated the expected cover size for pure configuration (see Section 3.3), given by Equation (3.5). To validate our theoretical result (Equation (3.5)), we ran the pure cover generation algorithm for all possible configurations of the set $\binom{n}{r}$ in a tree with 32 leaf nodes. The experimental results for the expected cover size match very closely with the values predicted by the expression (Equation 3.5), confirming the tightness of our expression.

Next, we repeat the same experiment with different database sizes for $n = 512$, 1024, 2048, and 4096. In this setting, it would be computationally prohibitive to compute the cover size of all configurations corresponding to a value of r, for all $0 \leq r \leq n$. Hence, for each r, we generate 5000 uniform random configurations (with replacement), then compute their mixed cover and the average cover size over these 5000 configurations. The results of this experiment are depicted in Figure 3-12, which shows the variation of the average cover size with r for different values of n. In each figure, the line $Y = X$ is also drawn for reference. For pure covers, the curves were drawn using Equation (3.5). The results in Figure 3-12 show the same pattern as in Figure 3-11.

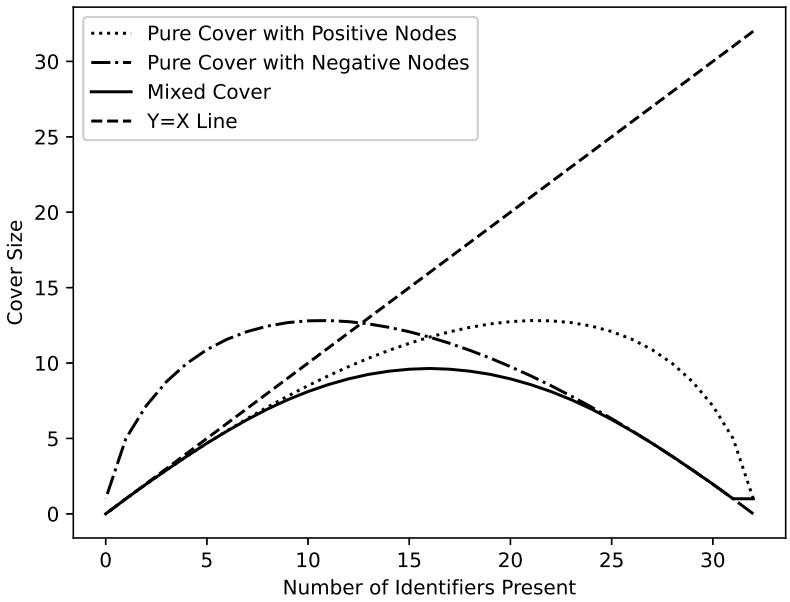
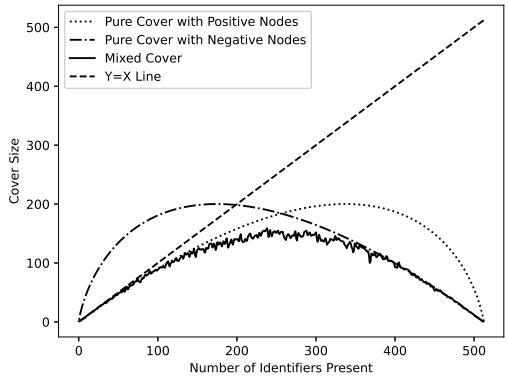


Figure 3-11: Number of Identifiers vs Average Cover Size

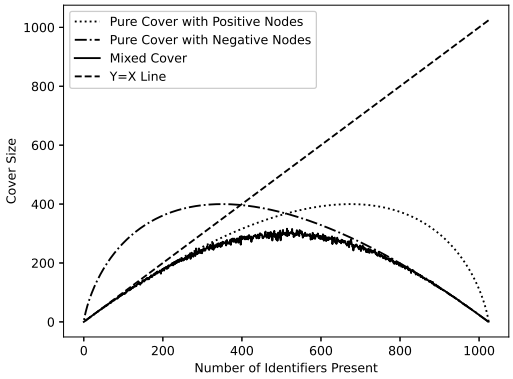
3.5 Final Remarks

We introduced the concept of a cover of a tree, which is a novel method for representing trees with labeled leaves more concisely. This representation has interesting combinatorial properties, which we explored in detail. We described several algorithms for generating covers and the corresponding algorithms for reconstructing the configuration of a tree. We also proved that our algorithm(s) produce optimal sized covers. We also provided an estimation of the expected size of a cover and experimentally validated our estimates.

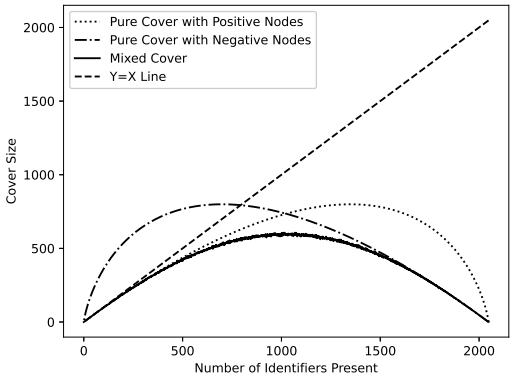
In the following chapter, we demonstrate how the techniques developed in this chapter can be applied to transform a generic SSE scheme into an equivalently secure and efficient version.



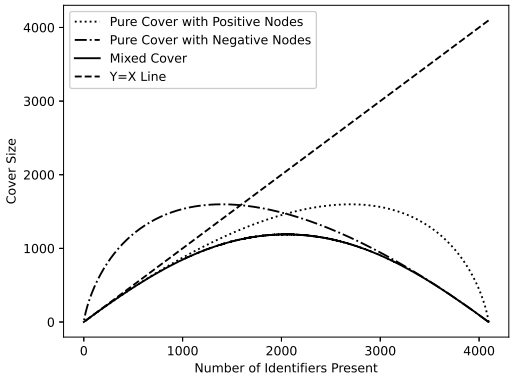
(a) Expected Cover Size (512)



(b) Expected Cover Size (1024)



(c) Expected Cover Size (2048)



(d) Expected Cover Size (4096)

Figure 3-12: Expected Cover Size Comparison

4 Tree Cover Based SSE

In this chapter, we show how to use our formulation of tree cover and the algorithms for cover generation discussed in Chapter 3 to design a cover based SSE scheme. Our method is generic, i.e., given any secure SSE scheme we can apply our method to convert it into a cover based scheme which will make the resulting scheme more efficient in terms of storage, search time and communication costs. This conversion of any base SSE scheme to cover based SSE scheme is security preserving, i.e., the new scheme retains the security of the base scheme.

To demonstrate our idea, we start by describing our technique for a static SSE in Section 4.1. Further in Section 4.2 we extend our scheme for dynamic SSEs. In Section 4.3, we describe in detail our methodology and point out how our scheme can result in savings if paired with an existing SSE. In Section 4.4, we also provide a detailed security argument showing that the new scheme resulting from pairing our scheme with a base SSE scheme enjoys the same security as the base scheme.

In Section 4.5 we validate our efficiency claims with extensive experimentation on both real and synthetic data. We report the results of our schemes when applied to the Enron Email database [45]. Our proposed scheme achieves a significant reduction (between 35% to 60%) in the size of the encrypted database over a base SSE scheme. We also simulate the dynamic setting in SSE using a synthetic database, and we demonstrate a significant advantage of our scheme in the dynamic setting as well. Furthermore, we provide results on extra overhead incurred by adopting our technique and show that the overhead of our scheme is fairly reasonable in a practical context.

The basic codes and data used for our experiment are publicly available at [77]. This chapter was partly published in [31].

4.1 Constructing Static SSE Using Tree Cover

A static database is where the number of documents in the database is a fixed natural number D. Let, $\mathcal{D} = \{\mathsf{d}_1, \ldots, \mathsf{d}_D\}$ be the set of documents and $\mathcal{I} = \{\mathsf{id}_1, \ldots, \mathsf{id}_D\}$ be the set of identifiers associated with those documents. We assume a natural ordering of the identifiers in the database, where id_i represents the i-th identifier of the database. Let $w \in \mathsf{W}$ be an arbitrary keyword and m_w be the largest integer such that $\mathsf{id}_{m_w} \in \mathsf{db}(w)$, and let h_w be the smallest integer such that $m_w \leq 2^{h_w}$.

Let $\mathcal{T}_w$ be a complete binary tree of height h_w. We will represent $\mathsf{db}(w)$ by a configuration of the tree $\mathcal{T}_w$. We associate each identifier id_i, $i \leq m_w$ with a leaf node of the tree $\mathcal{T}_w$, through the injective map $\varphi^{-1} : \mathcal{I} \to \mathsf{nodes}(\mathcal{T}_w)$, where

$$\varphi^{-1}(\mathsf{id}_i) = \phi^{-1}(i) = i + 2^{h_w} - 2.$$

Note, the ϕ^{-1} function was introduced in Section 3.1.1. With the above specification, $\varphi^{-1}(\mathsf{id}_i)$ represents the i^{th} leaf node from the left of the tree $\mathcal{T}_w$.

We label the leaf nodes of $\mathcal{T}_w$ as follows. For every $i \leq m_w$, the leaf node $\varphi^{-1}(\mathsf{id}_i)$ is labeled $+$ if $\mathsf{id}_i \in \mathsf{db}(w)$ and is labeled $-$ if $\mathsf{id}_i \notin \mathsf{db}(w)$. Moreover, if $m_w < 2^{h_w}$, then for all i, $m_w < i \leq 2^{h_w}$, the leaf nodes $\varphi^{-1}(\mathsf{id}_i)$ are labeled $-$. This labelling of the leaves of $\mathcal{T}_w$ yields a complete configuration of $\mathcal{T}_w$, and we call this configuration L_w. Note that this configuration L_w uniquely represents the set $\mathsf{db}(w)$.

We now fix an efficient tree cover scheme $\Psi = (\mathsf{CoverGen}, \mathsf{ReConstruct})$, and let $C_w \leftarrow \Psi.\mathsf{CoverGen}(h_w, L_w)$, where C_w be the cover of the configuration L_w of the tree $\mathcal{T}_w$ of height h_w. Thus, from C_w, the configuration L_w and further the set $\mathsf{db}(w)$ can be uniquely reconstructed. This interpretation of $\mathsf{db}(w)$ as a configuration of $\mathcal{T}_w$, which can be represented by a cover, will help us construct an efficient SSE scheme. It is important to note that the height of a tree completely specifies it. It is not

required by any of our schemes described later to store the tree explicitly.

4.1.1 Cover-based Representation of DB.

As a first step of constructing a cover-based SSE, we need to convert the given database DB to a different representation $\widetilde{\mathsf{DB}}$. We call this the *converted database*. This $\widetilde{\mathsf{DB}}$ then acts as the input to the existing SSE scheme. For each $w \in \mathsf{W}$, we generate a configuration L_w from the set $\mathsf{db}(w)$ as described before. Let $C_w \leftarrow \Psi.\mathsf{CoverGen}(L_w)$. We define

$$\widetilde{\mathsf{db}}(w) = \{(c, s, h_w) : (c, s) \in C_w\}. \tag{4.1}$$

The elements of $\widetilde{\mathsf{db}}(w)$ are the λ-bit encoding of the tuple (c, s, h_w). With this we define

$$\widetilde{\mathsf{DB}} = \bigcup_{w \in \mathsf{W}} \widetilde{\mathsf{db}}(w) \times \{w\}.$$

The conversion scheme from DB to $\widetilde{\mathsf{DB}}$ is summarized in $\mathsf{dbConversion}$ Algorithm in Figure 4-1.

$\mathsf{dbConversion}\,(\mathsf{DB})$	$\mathsf{Conversion}\,(\mathsf{db}(w))$
01. set $\widetilde{\mathsf{DB}} \leftarrow \emptyset$ 02. **for** each keyword $w \in \mathsf{W}$ 03. $\quad\widetilde{\mathsf{db}}(w) \leftarrow \mathsf{Conversion}(\mathsf{db}(w))$ 04. $\quad$**for** all $\widetilde{\mathsf{id}} \in \widetilde{\mathsf{db}}(w)$ 05. $\quad\quad\widetilde{\mathsf{DB}} \leftarrow \widetilde{\mathsf{DB}} \cup \{(\widetilde{\mathsf{id}}, w)\}$ 06. **return** $\widetilde{\mathsf{DB}}$	01. set $\widetilde{\mathsf{db}}(w), L_w \leftarrow \emptyset$ 02. let m_w be the largest integer such that $\mathsf{id}_{m_w} \in \mathsf{db}(w)$ 03. let h_w be the smallest integer such that $m_w \leq 2^{h_w}$ 04. initialize an empty full binary tree $\mathcal{T}_w$ of height h_w 05. define $\mathcal{P} = \{\varphi^{-1}(\mathsf{id}_i) : \mathsf{id}_i \in \mathsf{db}(w)\}$ 06. define $\mathcal{R} = \mathsf{leaves}(\mathcal{T}_w) \setminus \mathcal{P}$ 07. **for** all $p \in \mathcal{P}$ 08. $\quad L_w \leftarrow L_w \cup \{(p, +)\}$ 09. **for** all $r \in \mathcal{R}$ 10. $\quad L_w \leftarrow L_w \cup \{(r, -)\}$ 11. $C_w \leftarrow \Psi.\mathsf{CoverGen}(h_w, L_w)$ 12. **for** all $(c, s) \in C_w$ 13. $\quad\widetilde{\mathsf{db}}(w) \leftarrow \widetilde{\mathsf{db}}(w) \cup \{(c, s, h_w)\}$ 14. **return** $\widetilde{\mathsf{db}}(w)$

Figure 4-1: DB to $\widetilde{\mathsf{DB}}$ conversion algorithm

4.1.2 Generic Static SSE Using Keyword Cover

Let us consider a database

$$\mathsf{DB} = \bigcup_{w \in \mathsf{W}} \{(\mathsf{id}, w) : \mathsf{id} \in \mathsf{db}(w)\},$$

and any secure static SSE scheme $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\mathsf{Search})$ as defined in Definition 2.3.1 (without the update protocol). Our goal is to covert Σ into a new SSE $s\Sigma = (s\Sigma.\mathsf{Setup}, s\Sigma.\mathsf{Search})$. The procedures $s\Sigma.\mathsf{Setup}$ and $s\Sigma.\mathsf{Search}$ are described in Figure 4-2.

$s\Sigma.\mathsf{Setup}$ takes in the database DB and a security parameter λ and outputs an encrypted database EDB, a key k and a client state σ_C. EDB is uploaded to the server and σ_C and k are retained with the client. $s\Sigma.\mathsf{Setup}$ calls the routine $\mathsf{dbConversion}(\mathsf{DB})$, described in Figure 4-1 and converts DB to $\widetilde{\mathsf{DB}}$. Further $\widetilde{\mathsf{DB}}$ is sent as input to $\Sigma.\mathsf{Setup}$, the setup routine of the base static SSE scheme.

Search for a keyword w is performed by running the $s\Sigma.\mathsf{Search}$ protocol described in Figure 4-2. In the protocol, first the client's side $\mathsf{Search}_\mathsf{C}$ algorithm corresponding to the base SSE scheme Σ is executed on the inputs $(k, \sigma_\mathsf{C}, w)$. $\mathsf{Search}_\mathsf{C}\ (k, \sigma_\mathsf{C}, w)$ returns a search token stk_w for the keyword w and the updated client state σ_C. The search token stk_w is sent to the server, which subsequently runs the $\Sigma.\mathsf{Search}_\mathsf{S}\ (\mathsf{stk}_w, \mathsf{EDB})$ and outputs res, the search result, which is sent to the client. The rest of the procedure, i.e., lines 15 to 23 of Figure 4-2, runs on the client side. Where the client obtains the set $\widetilde{\mathsf{db}}(w)$ from the search result sent by the server. Note, as described in Equation 4.1, $\widetilde{\mathsf{db}}(w)$ is a collection of tuples which encode a cover of a tree of height h_w. Using the cover reconstruction algorithm, we reconstruct the configuration of the tree representing the set $\mathsf{db}(w)$ and finally output it.

It is worth noting a few important characteristics of the scheme $s\Sigma$.

Correctness. The correctness of $s\Sigma$ directly follows from the correctness of the base static SSE scheme Σ and the correctness of the cover generation scheme Ψ.

$s\Sigma.\mathsf{Setup}(1^\lambda, \mathsf{DB})$	$s\Sigma.\mathsf{Search}(k, \sigma_\mathsf{C}, w)$
01. $\widetilde{\mathsf{DB}} \leftarrow \mathsf{dbConversion}(\mathsf{DB})$ 02. $(k, \sigma_\mathsf{C}, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Setup}_\mathsf{C}(1^\lambda, \widetilde{\mathsf{DB}})$ 03. Send EDB to server and retain (k, σ_C)	01. Generate $(\sigma_\mathsf{C}, \mathsf{stk}_w) \leftarrow \Sigma.\mathsf{Search}_\mathsf{C}(k, \sigma_\mathsf{C}, w)$ 02. Send stk_w to the server 03. $\mathsf{res} \leftarrow \Sigma.\mathsf{Search}_\mathsf{S}(\mathsf{stk}_w, \mathsf{EDB})$ 04. Send res to the client 05. Decrypt and generate $\widetilde{\mathsf{db}}(w)$ from res 06. Set $C_w, \mathsf{db}(w) \leftarrow \emptyset$ 07. **for** each $\widetilde{\mathsf{id}} \in \widetilde{\mathsf{db}}(w)$ 08. Parse $\widetilde{\mathsf{id}}$ as (c, s, h_w) 09. $C_w \leftarrow C_w \cup \{(c, s)\}$ 10. Generate $L_w \leftarrow \Psi.\mathsf{ReConstruct}(h_w, C_w)$ 11. **for** each $(\ell, s) \in L_w$ 12. **if** $s = +$ 13. $\mathsf{db}(w) \leftarrow \mathsf{db}(w) \cup \{\varphi(\ell)\}$ 13. **return** $\mathsf{db}(w)$ to client

Figure 4-2: Generic static SSE using tree-cover scheme

Benefits. Performance of any SSE schemes is measured by the search and update time and the amount of data communicated during these processes. All these parameters essentially depend on the databases under consideration. For a keyword w, let $n_w = |\mathsf{db}(w)|$ be the number of document identifiers containing the keyword w. In all the state-of-the-art SSE schemes, either static [42] or dynamic, the search complexity is $\mathcal{O}(n_w)$ [88, 43, 33]. In many dynamic schemes, like in [60, 28, 23, 25, 38], the search complexity is in order of the number of updates for a keyword, which in the worst case may exceed $\mathcal{O}(n_w)$. Asymptotically, the worst-case search complexity of our scheme is also $\mathcal{O}(n_w)$, but on average the exact number of tuples that need to be communicated is much less than n_w. To the best of our knowledge, in no existing scheme the search complexity is smaller than n_w.

Note that, this improvement of search time is obtained by our scheme because we use a compact but lossless representation of $\mathsf{db}(w)$. This compact representation results in a smaller index size which further results in more efficient search and communication. The exact savings obtained by pairing our scheme with existing SSEs are further discussed in Section 4.3. Concrete experimental data on real databases is presented in Section 4.5.

Extra overhead. Superficially, it may look that $s\Sigma$ requires more computation than Σ. As $s\Sigma$ requires the conversion of DB to $\widetilde{\mathsf{DB}}$ in the setup phase and the conversion of the cover to the configuration (lines 15-23 in Figure 4-2). These extra computations are negligible and take place on the client side. Conversion of db to $\widetilde{\mathsf{db}}$ for a database of decently large size only takes a few seconds whereas reconstruction of db from cover takes less than a second. More detailed discussion on this can be found in Section 4.5.1. The important savings that are achieved through $s\Sigma$ is that, on average, the size of $\widetilde{\mathsf{DB}}$ is much smaller than DB, and this will significantly reduce the costs of the routines $\Sigma.\mathsf{Search}$. Moreover, this will lead to a smaller size of res which leads to a lower communication cost.

Security. The scheme $s\Sigma$ just does some pre-processing of the input to the base scheme Σ and further does some post-processing of the decrypted output. These pre and post-processing take place on the client side and do not require any computation involving the secret key(s). This implies that the security of the base scheme Σ implies the security of $s\Sigma$.

$\mathsf{BufferUpdate}((\mathsf{op},\mathsf{in}),\mathsf{B})$:

01. **if** $(\mathsf{op} = \mathsf{add}) \wedge ((\mathsf{del},\mathsf{in}) \in \mathsf{B})$
02. $\quad \mathsf{B} \leftarrow \mathsf{B} \setminus \{(\mathsf{del},\mathsf{in})\}$
03. $\quad \mathsf{B} \leftarrow \mathsf{B} \cup \{(\mathsf{op},\mathsf{in})\}$
04. **else-if** $(\mathsf{op} = \mathsf{del}) \wedge ((\mathsf{add},\mathsf{in}) \in \mathsf{B})$
05. $\quad \mathsf{B} \leftarrow \mathsf{B} \setminus \{(\mathsf{add},\mathsf{in})\}$
06. $\quad \mathsf{B} \leftarrow \mathsf{B} \cup \{(\mathsf{op},\mathsf{in})\}$
07. **else**
08. $\quad \mathsf{B} \leftarrow \mathsf{B} \cup \{(\mathsf{op},\mathsf{in})\}$
09. **return** B

Figure 4-3: Buffer update algorithm

We make this intuition more concrete in Section 4.4, where we present a reductionist security proof of our dynamic scheme $d\Sigma$ (described later in Section 4.2). The security result and the proof are also applicable to the static scheme.

Additional security advantage. Informally, an SSE scheme is called *volume hiding* if an adversary cannot guess the number of tuples that are related to a query by

seeing the result of a query which is transmitted by a server. This additional security property in SSE schemes has been recently studied [79, 59, 7]. In general, making an SSE volume hiding makes it inefficient in terms of communication and storage costs. In our scheme, the size of a search result for a keyword w is related to the size of the cover of the keyword w, and it does not directly reveal the number of tuples related to the keyword w. Thus, a constrained passive adversary, who only sees the communication between the server and the client, will not be able to accurately estimate the size of a query result from observing the response sizes. We believe that by using some additional randomness we can make our scheme to be volume hiding for more powerful adversaries.

$d\Sigma.\mathsf{Setup}\left(1^\lambda, \mathsf{DB}\right)$:

01. Set $\mathsf{B} \leftarrow \emptyset$
02. Initialize an empty map ts
03. $\widetilde{\mathsf{DB}} \leftarrow \mathsf{dbConversion}_d(\mathsf{ts}, \mathsf{DB})$
04. Run $(k, \sigma_\mathsf{C}, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Setup}_\mathsf{C}(1^\lambda, \widetilde{\mathsf{DB}})$
05. Send EDB to server and retain $(k, \sigma_\mathsf{C}, \mathsf{B}, \mathsf{ts})$

$\mathsf{dbConversion}_d\,(\mathsf{ts}, \mathsf{DB})$

01. set $\widetilde{\mathsf{DB}} \leftarrow \emptyset$
02. **for** all $w \in \mathcal{W}$
03. $\quad \mathsf{ts}_w \leftarrow \perp$
02. **for** each keyword $w \in \mathsf{W}$
03. $\quad$ Set $\mathsf{ts}_w \leftarrow 0$
04. $\quad \mathsf{DB}^* \leftarrow \mathsf{dbConversion}\,(\mathsf{DB})$
05. $\quad$ **for** all $(\mathsf{id}, w) \in \mathsf{DB}^*$
06. $\quad\quad \widetilde{\mathsf{id}} \leftarrow \mathsf{id} \| \mathsf{ts}_w$
07. $\quad\quad \widetilde{\mathsf{DB}} \leftarrow \widetilde{\mathsf{DB}} \cup \{(\widetilde{\mathsf{id}}, w)\}$
08. **return** $\widetilde{\mathsf{DB}}$

Figure 4-4: A generic dynamic SSE using tree-cover scheme setup phase and DB to $\widetilde{\mathsf{DB}}$ conversion algorithm

4.2 Constructing Dynamic SSE Using Tree Cover

For constructing a dynamic SSE, we need to support modifications in the database. Modification may take place in two ways. One being the addition of new documents in the database and the other being updating an existing document. In the context of SSE, a modification to the database is recorded by adding or removing the corresponding keyword identifier pair involved in the modification. In the context of the tree cover-based SSE scheme, we will still represent $\mathsf{db}(w)$ as a tree, and thus adding a new document would result in adding an extra leaf node to the existing tree, which

in some cases can only be done by increasing the height of the tree. Modification of an existing document would be achieved by assigning or altering the sign of the leaf node associated with the identifiers that were affected by the update operation.

As before we take a secure dynamic SSE scheme $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\mathsf{Search}, \Sigma.\mathsf{Update})$ and convert it into a tree cover based dynamic SSE $d\Sigma = (d\Sigma.\mathsf{Setup}, d\Sigma.\mathsf{Search}, d\Sigma.\mathsf{Update})$. The procedures for $d\Sigma.\mathsf{Setup}$ are shown in Figures 4-4. $d\Sigma.\mathsf{Update}$ is shown in Figures 4-3 and 4-5. $d\Sigma.\mathsf{Search}$ is shown in Figure 4-6.

$d\Sigma.\mathsf{Setup}$ described in Figure 4-4 is very similar to $s\Sigma.\mathsf{Setup}$. In $s\Sigma.\mathsf{Setup}$ a given database DB is converted to $\widetilde{\mathsf{DB}}$, where $\widetilde{\mathsf{DB}}$ consists of tuples of the form (w, c, h_w) where c is an element of the cover corresponding to the keyword w and h_w is the height of the corresponding tree representing the set $\mathsf{db}(w)$. In the case of $d\Sigma.\mathsf{Setup}$ the initial database, DB is converted into $\widetilde{\mathsf{DB}}$, but in this case, the $\widetilde{\mathsf{DB}}$ consists of tuples of the form $(w, c, h_w, \mathsf{ts}_w)$, where ts_w is a new variable associated with each keyword, which keeps information about the time at which some identifiers related to w have been updated. Further, we'll call ts_w as the time stamp for the keyword w.

In the setup phase, for each keyword w, ts_w is set to zero, signifying that no update has taken place yet. The role of this variable ts_w will be more clear from the update operation which we describe next.

Updates, in our case, take place by adding or deleting keyword-identifier pairs. Our main strategy for the update is a *lazy update* model. We assume a buffer memory B of restricted size (which may be user defined) at the client's side to store intermediate updates. The client uses this buffer B to store a "few" keyword-identifier pairs in un-encrypted form along with the corresponding operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$, where add and del represents addition and deletion respectively. Once the buffer is full, the client uploads the contents of the buffer to the server and resets the buffer to empty. The update procedure consists of procedures to update the buffer and procedures to upload the contents of the buffer to the server. The update procedure is summarized in Figure 4-3.

The update procedure shown in Figure 4-5 takes as input $(\mathsf{op}, \mathsf{in})$, where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ and in is a keyword-identifier pair (id, w). On receiving the input the client

$d\Sigma.\mathsf{Update}(k, \sigma_\mathsf{C}, \mathsf{ts}, \mathsf{B}, (\mathsf{op}, \mathsf{in}))$:

Client Side:

01. **if** $\mathsf{B} \neq \mathsf{FULL}$
02. $\quad \mathsf{B} \leftarrow \mathsf{BufferUpdate}(\mathsf{op}, \mathsf{in}, \mathsf{B})$
03. **if** $\mathsf{B} = \mathsf{FULL}$
04. $\quad$ set $\mathsf{UList} \leftarrow \emptyset$
05. $\quad$ **for** each w such that $(\mathsf{op}, (\mathsf{id}, w)) \in \mathsf{B}$
06. $\quad\quad$ set $L_w \leftarrow \emptyset$
07. $\quad\quad$ let m_w be the largest integer such that $(\mathsf{op}, (\mathsf{id}_{m_w}, w)) \in \mathsf{B}$
08. $\quad\quad$ let h_w be the smallest integer such that $m_w \leq 2^{h_w}$
09. $\quad\quad$ initialize an empty full binary tree $\mathcal{T}_w$ of height h_w
10. $\quad\quad$ define $\mathcal{P} = \{\varphi^{-1}(\mathsf{id}) : (\mathsf{add}, (\mathsf{id}, w)) \in \mathsf{B}\}$
11. $\quad\quad$ define $\mathcal{R} = \{\varphi^{-1}(\mathsf{id}) : (\mathsf{del}, (\mathsf{id}, w)) \in \mathsf{B}\}$
12. $\quad\quad$ **if** $\mathsf{ts}_w = \perp$
13. $\quad\quad\quad$ $\mathsf{ts}_w \leftarrow 0$
14. $\quad\quad\quad$ **for** all $p \in \mathcal{P}$
15. $\quad\quad\quad\quad$ $L_w \leftarrow L_w \cup \{(p, +)\}$
16. $\quad\quad\quad$ **for** all $r \in \mathsf{leaves}(\mathcal{T}_w) \setminus \mathcal{P}$
17. $\quad\quad\quad\quad$ $L_w \leftarrow L_w \cup \{(r, -)\}$
18. $\quad\quad\quad$ $(h_w, C_w) \leftarrow \Psi_d.\mathsf{CoverGen}(L_w)$
19. $\quad\quad$ **else**
20. $\quad\quad\quad$ $\mathsf{ts}_w \leftarrow \mathsf{ts}_w + 1$
21. $\quad\quad\quad$ **for** all $p \in \mathcal{P}$
22. $\quad\quad\quad\quad$ $L_w \leftarrow L_w \cup \{(p, +)\}$
23. $\quad\quad\quad$ **for** all $r \in \mathcal{R}$
24. $\quad\quad\quad$ $L_w \leftarrow L_w \cup \{(r, -)\}$
25. $\quad\quad$ $C_w \leftarrow \Psi_d.\mathsf{CoverGen}(h_w, L_w)$
26. $\quad\quad$ **for** each $(c, s) \in C_w$
27. $\quad\quad\quad$ $\widetilde{\mathsf{id}} = (c, s, h_w, \mathsf{ts}_w)$
28. $\quad\quad\quad$ $(\sigma_\mathsf{C}, \mathsf{utk}) \leftarrow \Sigma.\mathsf{Update}_\mathsf{C}\left(k, \sigma_\mathsf{C}, \mathsf{add}, (\widetilde{\mathsf{id}}, w)\right)$
29. $\quad\quad\quad$ $\mathsf{UList} \leftarrow \mathsf{UList} \cup \{\mathsf{utk}\}$
30. $\quad$ **send** UList to server

Server Side:

31. **for** all $\mathsf{utk} \in \mathsf{UList}$
32. $\quad \mathsf{EDB} \leftarrow \Sigma.\mathsf{Update}_\mathsf{S}(\mathsf{utk}, \mathsf{EDB})$

Figure 4-5: A generic dynamic SSE using tree-cover scheme update phase

first checks if $(\overline{\mathsf{op}}, \mathsf{in}) \in \mathsf{B}$ or not, where $\overline{\mathsf{op}} = \mathsf{del}$ if $\mathsf{op} = \mathsf{add}$ and vice versa. If $(\overline{\mathsf{op}}, \mathsf{in}) \in \mathsf{B}$, then the client deletes $(\overline{\mathsf{op}}, \mathsf{in})$ from B and add $(\mathsf{op}, \mathsf{in})$ to the buffer B. Otherwise, it only adds $(\mathsf{op}, \mathsf{in})$ to the buffer B. This procedure is summarized in the procedure $\mathsf{BufferUpdate}$ shown in Figure 4-3. It is important to note that if $(\overline{\mathsf{op}}, \mathsf{in})$ is present in the buffer, then only deleting $(\overline{\mathsf{op}}, \mathsf{in})$ from the buffer does not suffice. It is also necessary to add $(\mathsf{op}, \mathsf{in})$ to the buffer B as the client has no knowledge of the current contents of the server, in particular, it is not possible for the client to know during the update process if $(\overline{\mathsf{op}}, \mathsf{in})$ is currently present in server or not. For example, assume that for an identifier id the client wants to delete a keyword w, i.e., $\mathsf{op} = \mathsf{del}$. Also assume that $(\mathsf{add}, (\mathsf{id}, w)) \in \mathsf{B}$. If (id, w) currently resides in the server, then deleting $(\mathsf{add}, (\mathsf{id}, w))$ from buffer and not adding $(\mathsf{del}, (\mathsf{id}, w))$ to B would lead to a wrong configuration for the keyword w.

After the buffer B becomes full the client retrieves all the entries $(\mathsf{op}, \mathsf{in}) \in \mathsf{B}$ that correspond to each keyword w. Let $\mathsf{Udt}(w) = \{(\mathsf{op}, \mathsf{in}) \in \mathsf{B} : \mathsf{in} = (\mathsf{id}, w)\}$. The set $\mathsf{Udt}(w)$ gives all identifiers corresponding to w which were updated in the current phase. The client creates a tree $\mathcal{T}$ whose leaf nodes based on the identifiers present in $\mathsf{Udt}(w)$, the identifiers associated with the operation add are labeled $+$ and the ones associated with del are labeled with $-$. This labeling gives a configuration L_w of the tree $\mathcal{T}$. This configuration along with the height of the tree is fed to a dynamic cover generation algorithm $\Psi_d.\mathsf{dCoverGen}$, which yields a cover C_w for the configuration L_w. C_w consists of pairs (i, s) where i is a node of the tree $\mathcal{T}$ and $s \in \{+, -\}$. The update operation uploads a λ bit representation of the tuple $(i, s, h_w, \mathsf{ts}_w)$ to the server by creating an update token for the string through the client side update procedure of the base SSE, i.e., through $\Sigma.\mathsf{Update}_\mathsf{C}$. The details are depicted in the Algorithm shown in Figure 4-5.

To perform a search query on w, the client uses the $\Sigma.\mathsf{Search}_\mathsf{C}$ protocol to search for the keyword w and obtains the search token stk_w. This token is then sent to the server. Upon receiving the search token, the server returns the encrypted search result $\mathsf{res}_\mathsf{S} \leftarrow \Sigma.\mathsf{Search}_\mathsf{S}(\mathsf{stk}, \mathsf{EDB})$, which the client decrypts. Each element of the decrypted search result res is an encoding of (c, s, h, ts_w). The client generates a

$d\Sigma.\mathsf{Search}\,(k, \sigma_\mathsf{C}, \mathsf{ts}, \mathsf{B}, w)$:

Round 1:

Generate $(\sigma_\mathsf{C}, \mathsf{stk}_w) \leftarrow \Sigma.\mathsf{Search}_\mathsf{C}\,(k, \sigma_\mathsf{C}, w)$
Send stk_w to server
Server returns $\mathsf{res}_\mathsf{S} \leftarrow \Sigma.\mathsf{Search}_\mathsf{S}\,(\mathsf{stk}, \mathsf{EDB})$
Parse all $r \in \mathsf{res}_\mathsf{S}$ as (c, s, h, i)
for $j \in [0, \mathsf{ts}_w - 1]$
    Initialize $C_j \leftarrow \emptyset$
    $C_j \leftarrow C_j \cup \{(c, s)\}$, and $h_j \leftarrow h$ for all $(c, s, h, j) \in \mathsf{res}_\mathsf{S}$
Generate the final configuration $L \leftarrow \Psi_d.\mathsf{dReConstruct}\left(\{h_j, C_j\}_{j \in [0, \mathsf{ts}_w - 1]}\right)$
Generate $\mathsf{db}(w)$ from L using φ
Search in B with keyword w and create the set res_C from the search result
for all $(\mathsf{op}, (\mathsf{id}, w)) \in \mathsf{res}_\mathsf{C}$
    if $\mathsf{op} = \mathsf{add}$
        $\mathsf{db}(w) \leftarrow \mathsf{db}(w) \cup \{\mathsf{id}\}$
    else
        $\mathsf{db}(w) \leftarrow \mathsf{db}(w) \setminus \{\mathsf{id}\}$

Round 2:

Client Side:

Set $\mathsf{UList} \leftarrow \emptyset$
Set $\mathsf{ts}_w \leftarrow 0$
$\widetilde{\mathsf{db}}(w) \leftarrow \mathsf{Conversion}(\mathsf{db}(w))$
for all $\mathsf{id} \in \widetilde{\mathsf{db}}(w)$
    $\widetilde{\mathsf{id}} \leftarrow (\mathsf{id}, \mathsf{ts}_w)$
    $(\sigma_\mathsf{C}, \mathsf{utk}) \leftarrow \Sigma.\mathsf{Update}_\mathsf{C}\left(k, \sigma_\mathsf{C}, \mathsf{add}, (\widetilde{\mathsf{id}}, w)\right)$
    $\mathsf{UList} \leftarrow \mathsf{UList} \cup \{\mathsf{utk}\}$
Send UList to server

Server Side:

for all $\mathsf{utk} \in \mathsf{UList}$
    $\mathsf{EDB} \leftarrow \Sigma.\mathsf{Update}_\mathsf{S}(\mathsf{utk}, \mathsf{EDB})$

Figure 4-6: A generic dynamic SSE using tree-cover scheme search phase

sequence of covers $\{(h_t, C_t)\}_{t \in [0, \mathsf{ts}_w - 1]}$, where

$$C_t = \{(c, s) : (c, s, h, \mathsf{ts}_w) \in \mathsf{res} \text{ and } , \mathsf{ts}_w = t\},$$

for all $t \in [0, \mathsf{ts}_w - 1]$. It is important to note that for every update (that is when the buffer is full and offloaded to the server), the timestamp and corresponding height h related to all the updates of a particular keyword are the same. The client then feeds this sequence of covers to the dynamic cover reconstruction algorithm $\Psi_d.\mathsf{dReConstruct}$

(Section 3.2.4). The final configuration produced by the Ψ_d.dReConstruct algorithm is used to construct $\mathsf{db}(w)$. For computing the final search result the client looks for w in the buffer B and denotes the search result as $\mathsf{res_C}$. The entries in B are in $\mathsf{res_C}$ and have the format $(\mathsf{op}, (\mathsf{id}, w))$. If $(\mathsf{del}, (\mathsf{id}, w)) \in \mathsf{res_C}$ then client discards the id from $\mathsf{db}(w)$. Otherwise, if $(\mathsf{add}, (\mathsf{id}, w)) \in \mathsf{res_C}$ then the client adds id to $\mathsf{db}(w)$. Subsequently, it resets ts_w to 0. The client then re-uploads the search result for w, in a manner similar to the update phase discussed earlier. The details are in Figure 4-6.

4.3 Discussions

In this section we discuss some existing SSEs and the consequences of pairing our scheme with them.

Consider a dynamic SSE scheme applied to an initially empty database. At a certain instance of time, let i_w be the number of additions, and d_w be the number of deletions that have taken place for a keyword w. Thus, the total number of updates u_w for the keyword w is given by $u_w = i_w + d_w$, and $n_w = i_w - d_w$ denote the number of identifier pairs currently matching keyword w. In addition, let N be the total number of document identifier pairs in the database at that instance. In Table 4.1 we summarize the characteristics of some widely studied recent SSE schemes. The list does not pretend to be a complete one but is a good representative of the existing SSE schemes. The Table has two major columns named **Computation Cost** and **Communication Cost**. The two sub-columns under Computation Cost report the asymptotic computation cost for search and update, respectively. The three sub-columns under Communication Cost list the size of the result of a single keyword search, the size of an update token for a single update and the number of round-trips (RT), i.e., the number of communication rounds necessary between the server and client for a search.

The schemes reported in Table 4.1 can be naturally grouped into two groups as follows:

Group-1: The schemes whose search cost is $\mathcal{O}(u_w)$. The first four entries of

Table 4.1, i.e., Fides, Mitra, Π_{BP} and $\mathsf{Diana}_{\mathsf{del}}$ falls under this category.

Group-2: The schemes whose search cost is not $\mathcal{O}(u_w)$ but is dominated by n_w. The last five entries of Table 4.1 fall under this group.

SSE protocols that achieve a $\mathcal{O}(n_w)$ time complexity for search are called optimal search protocols. The *Group-2* schemes are near-optimal, as the leading term in the search cost of these schemes is dominated by n_w.

The *Group-1* schemes fail to achieve the optimal search complexity as they treat deletion also as an insertion with a specific tag and thus the search time depends on the number of updates and not on the number of documents currently in the database that contains w. But the *Group-2* schemes achieve near-optimal search time at an increased cost for updates. All *Group-2* schemes except Janus and LLSE have an update cost of $\mathcal{O}(\log N)$ whereas most *Group-1* schemes have a constant update cost.

A similar pattern is observed in the case of communication costs. The size of the search results of all schemes in *Group-2* is dominated by n_w, but this is achieved with an increased number of communication rounds. Most *Group-1* schemes require a small constant number of communication rounds, but the response size for a search query is $\mathcal{O}(u_w)$.

Effect of our Pre-processing: Our proposed scheme is just a pre-processing step which can be applied to all existing SSEs. For a concrete understanding, we can consider the effect of our pre-processing step on the schemes listed in Table 4.1. Firstly, if the tree cover scheme is paired with any of the listed schemes the asymptotic complexity of the schemes does not change. For each w, our scheme deals with $\widetilde{\mathsf{db}}(w)$ instead of $\mathsf{db}(w)$, hence the parameters of interest on which the complexity is measured in all the listed schemes will change. In particular, with our scheme the parameter $n_w = |\mathsf{db}(w)|$ should be replaced by $\widetilde{n}_w = |\widetilde{\mathsf{db}}(w)|$ and $N = |\mathsf{DB}|$ should be replaced by $\widetilde{N} = |\widetilde{\mathsf{DB}}|$. We have already amply argued that on average for any database we will have $\widetilde{n}_w < n_w$ and $\widetilde{N} < N$. With this, it is easy to see that on average each of the *Group-2* schemes will have a concrete reduction of both computation and communication costs.

The effect of our scheme in the case of the *Group-1* schemes is similar. In these schemes, the search cost grows linearly with the number of updates u_w. These schemes consider each update, either insertion or deletion, as a new keyword document pair and these are stored in the database. Thus, the number of keyword document pairs currently in the database is u_w, and this leads to the linear dependence of the search time with the number of updates. But, if paired with the tree cover scheme, the effective number of updates that are to be stored will get reduced on average as instead of the keyword identifier pairs the cover of the configuration related to those pairs will be stored and this would incur a lesser cost. Based on the same argument, there would be a concrete reduction of the response sizes on average. Moreover, the constant update cost and the constant update token sizes, as achieved by these schemes, would be retained if paired with the tree cover scheme.

However, our pre-processing incurs some additional overhead on the client, as discussed in detail in Section 3.2.3 and experimentally analyzed in Section 4.5.1 (specifically in Table 4.3). These results demonstrate that the pre-processing time remains reasonably low for adequately large databases and provides significant benefits in terms of server storage, search efficiency, and communication overhead.

The use of the buffer memory also adds to the overall performance of the scheme. However, the buffer memory is typically very small as compared to the size of the database. While the buffer does incur additional search time, which is linear in the size of the buffer, it significantly reduces communication overhead. Without the buffer, the update protocol would need to be triggered immediately whenever an update occurs. Additionally, the buffer allows the SSE to take advantage of our proposed tree cover scheme.

4.4 Security of $d\Sigma$

As already stated, our scheme $d\Sigma$ acts as a key-less pre-processing step on a base SSE scheme Λ. Thus, if our scheme is used as a pre-processing over an SSE Λ, the resulting scheme would inherit the security of Λ. In this section, we formalize this

Scheme	Computation Cost		Communication Cost		
	Search	Update	Search	Update	RT
Fides [25]	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	2
Mitra [49]	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	1
Π_{BP} [38]	$\mathcal{O}(u_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	2
$\mathsf{Diana}_{\mathsf{del}}$ [25]	$\mathcal{O}(u_w)$	$\mathcal{O}(\log u_w)$	$\mathcal{O}(n_w + d_w \log u_w)$	$\mathcal{O}(1)$	2
Janus [25]	$\mathcal{O}(n_w \cdot d_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	1
QOS [43]	$\mathcal{O}(n_w \log i_w + \log^2 \lvert\mathcal{W}\rvert)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^3 N)$	$\mathcal{O}(\log \lvert\mathcal{W}\rvert)$
Orion [49]	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$
Horus [49]	$\mathcal{O}(n_w \log(d_w) \log N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w \log^2 N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(\log N)$
LLSE [33]	$\mathcal{O}((n_w + \log i_w) \cdot \log \log N)$	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(n_w)$	$\mathcal{O}(\log^2 N)$	1

Table 4.1: Characteristics of some existing SSE schemes: N is the number of (id, w) pairs, $|\mathcal{W}|$ is the number of keywords, i_w and d_w are the number of insertions and deletions, and $u_w = i_w + d_w$ is the total number of updates, and $n_w = i_w - d_w$ is the number of keyword-identifier pairs currently matching the keyword w. RT is the number of roundtrips for a search query.

intuition.

The security of a Dynamic SSE Λ is determined by a leakage function $\mathcal{L} = (\mathcal{L}_{\mathsf{Setup}}, \mathcal{L}_{\mathsf{Search}}, \mathcal{L}_{\mathsf{Update}})$. $\mathcal{L}$ denotes the information that the adversary learns from the setup process and each execution of the search and update protocols. The security of SSE schemes is generally argued by showing that an adversary cannot distinguish between a real-world execution and an ideal-world execution (simulated using the leakage function) of the scheme [42, 60, 28].

We reduce the security of our dynamic scheme $d\Sigma$ to the security of the base SSE scheme Λ.

Theorem 4.4.1. *Let* $d\Sigma = (d\Sigma.\mathsf{Setup}, d\Sigma.\mathsf{Search}, d\Sigma.\mathsf{Update})$ *be a Dynamic SSE scheme as described in Figures 4-4, 4-5 and 4-6.* $d\Sigma$ *is instantiated with a fixed but arbitrary tree cover scheme* $\Psi = (\Psi.\mathsf{CoverGen}, \Psi.\ \mathsf{ReConstruct})$ *and a base dynamic SSE scheme* Λ *with leakage profile* $\mathcal{L}$*. If* Λ *is* $\mathcal{L}$*-adaptive secure, then* $d\Sigma$ *is also* $\mathcal{L}$*-adaptive secure.*

Proof. We say that $\mathsf{Sim}_{\mathcal{B}}$ is a *compatible simulator* for an adversary $\mathcal{B}$ attacking Λ if

$$\left|\Pr\left[\mathsf{SSEReal}^{\Lambda}_{\mathcal{B}}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Lambda}_{\mathcal{B},\mathsf{Sim}_{\mathcal{B}}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

As Λ is $\mathcal{L}$-adaptive secure hence by Definition 2.5.1, a compatible simulator for $\mathcal{B}$ always exists.

Let $\mathcal{A}$ be an arbitrary adversary for $d\Sigma$ instantiated with a fixed but arbitrary tree cover scheme Ψ and a base dynamic SSE Λ. Based on the protocols involved in $d\Sigma$ (see Figures 4-4, 4-5, 4-6) it is important to note down the correct interface between the adversary $\mathcal{A}$ and its challenger:

1. On a setup query $(1^\lambda, \mathsf{DB})$ for a database DB of $\mathcal{A}$'s choice its challenger returns EDB.

2. On an update query $(\mathsf{op}, \mathsf{in})$, its challenger returns UList a set of update tokens (see line 30 of Figure 4-5).

3. For a search query w, the challenger returns a search token stoken_w and a set of update tokens UList (See lines 02 and 23 of Figure 4-6).

Given an adversary $\mathcal{A}$ for $d\Sigma$, we construct an adversary $\mathcal{B}$ for the base protocol Λ, which acts as a challenger for $\mathcal{A}$. $\mathcal{B}$ being an adversary for Λ has a challenger which we denote as $\mathcal{C}$. $\mathcal{B}$ provides responses to the queries of $\mathcal{A}$ with the help of the responses it receives from its challenger $\mathcal{C}$. Note the tree cover scheme Ψ is key-less and is thus accessible to $\mathcal{B}$.

Now we describe the interaction of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ in a sequence of games $\mathsf{G0}$, $\mathsf{G1}$, $\mathsf{G2}$.

Game $\mathsf{G0}$: We assume that $\mathcal{B}$'s challenger $\mathcal{C}$ follows the real protocols $(\Lambda.\mathsf{Setup}, \Lambda.\mathsf{Search}, \Lambda.\mathsf{Update})$ to answer queries of $\mathcal{B}$. $\mathcal{B}$ acts as a challenger to $\mathcal{A}$ as follows:

1. When a setup request $\mathsf{Setup}(1^\lambda, \mathsf{DB})$ is issued by $\mathcal{A}$ then $\mathcal{B}$ runs lines 01 to 03 of the procedure $d\Sigma.\mathsf{Setup}(1^\lambda, \mathsf{DB})$ described in Figure 4-4 and obtains $\widetilde{\mathsf{DB}}$. $\mathcal{B}$ then issues the setup request $(1^\lambda, \widetilde{\mathsf{DB}})$ to its challenger $\mathcal{C}$ and receives EDB as a response which it transmits to $\mathcal{A}$. Note, in this process $\mathcal{B}$ has created a map ts and a buffer B which it retains with it and updates during the subsequent queries of $\mathcal{A}$.

2. On receiving an update query $(\mathsf{op}, \mathsf{in})$ from $\mathcal{A}$, $\mathcal{B}$ runs lines 01 to 30 of the procedure $d\Sigma.\mathsf{Update}$ described in Figure 4-5. In lieu of line 28, it issues a search

query $(\widetilde{\mathsf{id}}, w)$ to its challenger $\mathcal{C}$ and receives utoken in response. It populates the set UList using the responses received from $\mathcal{C}$, then it executes lines 31 and 32 of the procedure $d\Sigma.\mathsf{Update}$ and finally sends UList to $\mathcal{A}$.

3. On receiving a search query w from $\mathcal{A}$, $\mathcal{B}$ queries $\mathcal{C}$ with w and receives as a response stoken_w. Then it executes lines 03 to 25 of the procedure $d\Sigma.\mathsf{Search}$ as described in Figure 4-6. Instead of executing line 21, it queries $\mathcal{C}$ with an update query $(\mathsf{add}, (\widetilde{\mathsf{id}}, w))$ and receives utoken as response. Finally, it sends UList as constructed in line 23 to $\mathcal{A}$.

After $\mathcal{A}$ stops querying, $\mathcal{A}$ outputs a bit $b \in \{0,1\}$, $\mathcal{B}$ and $\mathsf{G0}$ also outputs b.

We can view $\mathsf{G0}$ as an interaction between $\mathcal{A}$ and its challenger $\mathcal{B}$, where $\mathcal{A}$ gets a perfect interface for the real scheme $d\Sigma$, thus

$$\Pr\left[\mathsf{SSEReal}_{\mathcal{A}}^{d\Sigma}(\lambda) = 1\right] = \Pr[\mathsf{G0} = 1]. \tag{4.2}$$

We can also view adversaries $\mathcal{A}$ and $\mathcal{B}$ together as a single adversary $\mathcal{B}'$ who interacts with $\mathcal{C}$ (the challenger of $\mathcal{B}$), which provides the perfect interface for Λ. Thus, we have

$$\Pr\left[\mathsf{SSEReal}_{\mathcal{B}'}^{\Lambda}(\lambda) = 1\right] = \Pr[\mathsf{G0} = 1]. \tag{4.3}$$

Game $\mathsf{G1}$: We make some small changes in Game $\mathsf{G0}$ to obtain game $\mathsf{G1}$. First, observe that for any adversary $\mathcal{A}$ for $d\Sigma$, Game $\mathsf{G0}$ gives a concrete description for an adversary $\mathcal{B}$, and thus of the combined adversary $\mathcal{B}'$ which attacks Λ. As Λ is $\mathcal{L}$-adaptive secure hence a compatible simulator $\mathsf{Sim}_{\mathcal{B}'}$ for $\mathcal{B}'$ exists. In Game $\mathsf{G1}$ the challenger responds to the queries of $\mathcal{B}'$ using the simulator $\mathsf{Sim}_{\mathcal{B}'}$ instead of the real protocols of Λ. Thus,

$$\Pr\left[\mathsf{SSEIdeal}_{\mathcal{B}',\mathsf{Sim}_{\mathcal{B}'}}^{\Lambda} = 1\right] = \Pr[\mathsf{G1} = 1]. \tag{4.4}$$

As Λ is $\mathcal{L}$-adaptive secure hence, we have

$$\Big|\Pr\left[\mathsf{SSEReal}^{\Lambda}_{\mathcal{B}'}(\lambda)=1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Lambda}_{\mathcal{B}',\mathsf{Sim}_{\mathcal{B}'}}(\lambda)=1\right]\Big| \leq \mathsf{negl}(\lambda). \quad (4.5)$$

Thus using Equations (4.3), (4.4, (4.5), we have

$$|\Pr[\mathsf{G0}=1] - \Pr[\mathsf{G1}=1]| \leq \mathsf{negl}(\lambda). \quad (4.6)$$

Game G2: In this game we view the game G1 a bit differently. We see the description of $\mathcal{B}$ along with the simulator $\mathsf{Sim}'_{\mathcal{B}}$ together and call it as $\mathsf{Sim}_{\mathcal{A}}$. Note, with this view, we see $\mathcal{A}$ interacting with a single piece of code $\mathsf{Sim}_{\mathcal{A}}$. Thus, we have

$$\Pr[\mathsf{SSEIdeal}^{\Lambda}_{\mathcal{A},\mathsf{Sim}_{\mathcal{A}}}(\lambda)] = \Pr[\mathsf{G2}=1], \quad (4.7)$$

and as G1 and G2 are essentially same we have

$$\Pr[\mathsf{G1}=1] = \Pr[\mathsf{G2}=1]. \quad (4.8)$$

Now, using Equations (4.6) and (4.8) we have

$$|\Pr[\mathsf{G0}=1] - \Pr[\mathsf{G2}=1]| \leq \mathsf{negl}(\lambda). \quad (4.9)$$

Finally, using Equations (4.2), (4.7) and (4.9), we have

$$\Big|\Pr\left[\mathsf{SSEReal}^{\Lambda}_{\mathcal{A}}(\lambda)=1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Lambda}_{\mathcal{A},\mathsf{Sim}_{\mathcal{A}}}(\lambda)=1\right]\Big| \leq \mathsf{negl}(\lambda),$$

as desired. □

A few more points regarding the security of $d\Sigma$ are to be noted:

1. Our pre-processing step does not add to the security of the base scheme. The tree cover scheme is meant to enhance the efficiency of the scheme while restoring the security of the base scheme.

2. Theorem 4.4.1 only asserts a basic security guarantee of our scheme. We assumed the base scheme to be $\mathcal{L}$-adaptive secure which is a well accepted model of security. But security of SSE schemes is still an active area of research and is not fully understood. It has been claimed that SSE schemes proven secure in the $\mathcal{L}$-adaptive model may still succumb to attacks that the security model fails to incorporate. For example, a class of attacks called file injection attacks [97] or a recent generalization in [6] may still be applicable to provably secure schemes. Such weaknesses of the base SSE scheme Λ may affect the security of $d\mathbf{\Sigma}$.

3. A similar security Theorem holds for our static scheme $s\mathbf{\Sigma}$.

4.5 Experimental Results

To validate our proposed scheme in practical databases, we conducted experiments using the Enron Email Dataset [45], which has about 500,000 documents and 200,000 keywords. We began by extracting keywords from the data set and construction of DB consisting of the keyword identifier pairs. Subsequently, using the algorithm in Figure 4-1 we converted DB to $\widetilde{\mathsf{DB}}$. We experimented on different sizes of the database by randomly selecting subsets of the database and for each case we compared the sizes of the original and converted versions. These results are shown in Table 4.2 and for easy visual comparison a pictorial representation of the data in Table 4.2 is shown in Figure 4-7. The last column of Table 4.2 computes $\frac{(|\mathsf{DB}|-|\widetilde{\mathsf{DB}}|)\times 100}{|\mathsf{DB}|}$. It is evident from Table 4.2 that our protocol results in substantial reductions in database size, ranging from 60% to 35%, even when dealing with reasonably large databases. This demonstrates the effectiveness of our approach in practical databases.

4.5.1 Extra Overheads

Our scheme builds over an existing SSE and the extra overhead over the original SSE is the time required for computing covers and cover reconstruction. We experimentally compute the time required for cover generation and computation. We used the

$\lvert\mathsf{DB}\rvert$	$\lvert\widetilde{\mathsf{DB}}\rvert$	Advantage (%)
2^{20}	5,26,492	52.4
$2^{20.5}$	6,08,347	60.2
2^{21}	9,24,455	54.8
$2^{21.5}$	15,50,264	49.8
2^{22}	29,74,256	43.8
$2^{22.5}$	44,05,133	41.1
2^{23}	64,47,481	36.4
$2^{23.5}$	75,17,379	35.6

Table 4.2: The size of original database DB and the converted database $\widetilde{\mathsf{DB}}$ in case of Enron data.

following configuration for our computation:

CPU: Intel Core i5-1135G7 @ 2.40GHz × 8 processor (3.1GHz).

RAM: 16 GB

OS: Ubuntu 22.04.3 LTS, 64 bits.

Programming Language: Python

Table 4.3 shows the time required for cover generation and reconstruction for databases of different sizes. We considered database sizes of $n = 2^{17}, 2^{18}, 2^{19}$. For each of these databases, we considered different sizes of $|\mathsf{db}(w)|$ as shown in the rows of the Table. For each $|\mathsf{db}(w)|$ we generated 100 random configurations and the times reported for the cover generation and re-construction are the average time required for generation and reconstruction for these 100 configurations. As expected, Table 4.3 clearly shows that the cover generation times increase with both the increase in n and $|\mathsf{db}(w)|$. The reconstruction times reported for $|\mathsf{db}(w)| < 10^4$, are negligible. If $|\mathsf{db}(w)|$ is small it is expected that a pure cover is generated, and reconstruction of a pure cover is immediate.

Dynamic Scenario: Now, we test the performance of our scheme for updates. We consider a database containing 10^9 keyword-identifier pairs, and a client buffer of size 0.01% of the actual database size. We consider updates for a single keyword w following the protocol described below.

We consider an initial set of documents $\mathcal{D}_0$ and a keyword w^* where w^* is initially

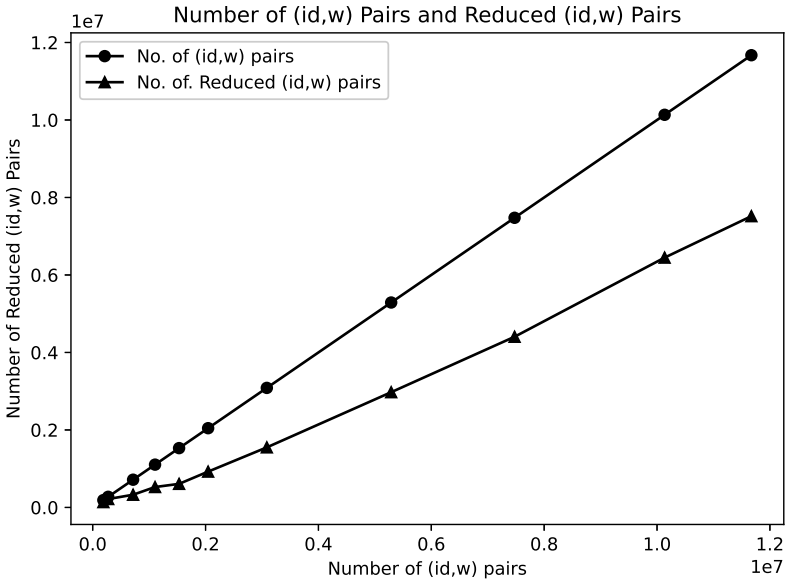


Figure 4-7: A pictorial depiction of the data in Table 4.2 showing the number of tuples in the original database (represented by filled circles) and the number of tuples in the converted database (represented by filled triangles) for different database sizes.

present in 12.5% of the documents in $\mathcal{D}_0$. Next, we perform a series of update operations involving w^*. The updates are made in three phases P_1, P_2, P_3. The updates in phase P_i are applied to the documents in $\mathcal{D}_{i-1}$, and the updates result in a new set of documents $\mathcal{D}_i$. The update operations are designed in such a way that w^* is present in 25% of the documents in $\mathcal{D}_1$, 50% of the documents in $\mathcal{D}_2$ and 60% documents in $\mathcal{D}_3$. In each phase, multiple update operations are performed and 10% of all operations in each phase are delete operations. At the end of each phase of updates, a search operation is performed involving w^*. Note, that the search operation forces

$\lvert\mathsf{db}(w)\rvert$	Time (sec) for $n = 2^{17}$		Time (sec) for $n = 2^{18}$		Time (sec) for $n = 2^{19}$	
	Gen	Re-con	Gen	Re-con	Gen	Re-con
10^2	0.22	10^{-5}	0.42	10^{-5}	0.94	10^{-5}
10^3	0.27	10^{-4}	0.49	10^{-4}	1.01	10^{-4}
10^4	0.51	10^{-3}	0.88	10^{-3}	1.51	10^{-3}
2^{16}	0.87	0.17	1.64	0.25	2.97	0.35
2^{17}	-	-	1.99	0.37	3.69	0.57
2^{18}	-	-	-	-	4.14	0.85

Table 4.3: Cover generation and reconstruction times.

the transmission of all tuples corresponding to w^* from the buffer to the server.

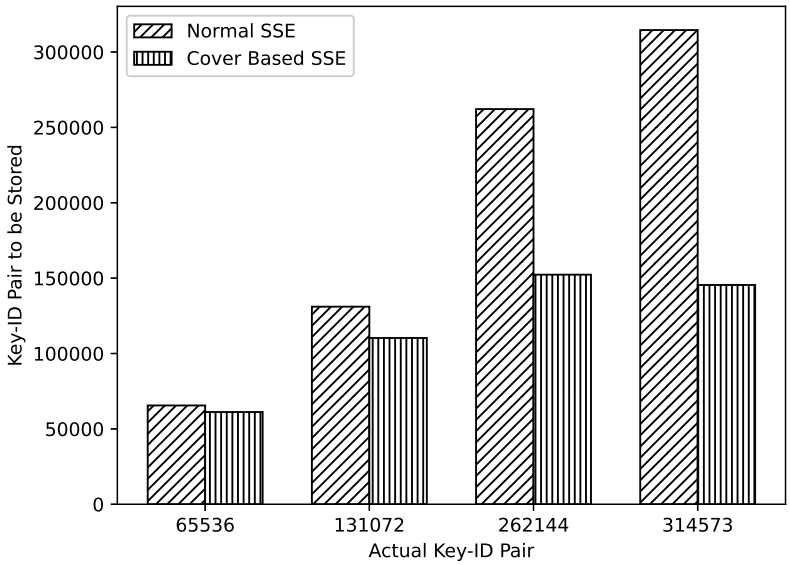


Figure 4-8: Database sizes in dynamic scenario.

We repeated the above protocol 2000 times with random updates following the rules stated above using our dynamic SSE scheme. After each execution, we recorded the size of the database after the search operations in each phase. In Figure 4-8 we report the average size of the database across the 2000 executions in a bar diagram. In Figure 4-8 the database sizes for our scheme and the base SSE are shown. The four pairs of bars correspond to the sizes of the databases after each phase just after the search operation. The first pair correspond to the database for the initial set of documents $\mathcal{D}_0$, the second pair is for $\mathcal{D}_1$ etc. The figure clearly shows that our scheme leads to considerable savings in the dynamic scenario.

4.6 Final Remarks

In this chapter we described a novel way to make SSE schemes space efficient. Our scheme converts a given database into a different representation which on average results in a significant amount of space savings. This smaller representation also results in reduced search time and response sizes. In our experiments, we explicitly show that our representation results in smaller index sizes with very low extra overhead. Our

scheme can be used with any secure SSE, and our scheme, being just a pre-processing step, retains the security of the base SSE.

5

Conjunctive Dynamic SSE – A Generic Framework

Most SSE schemes to date [42, 60, 28, 23, 87, 25, 49, 89, 38, 90] support single keyword searches per query. Complex searches may involve multiple keywords combined via conjunctions or general boolean expressions. A naive approach to achieve this is by performing individual single keyword searches and then computing set operations on the results. However, this method is inefficient and increases computation and communication between the client and server. This also reveals the number of matching documents for each keyword to the server.

There are a few SSE schemes that support conjunctive queries [29, 47, 2, 93, 58, 66, 80, 102, 78, 100, 95]. The most notable of these is the oblivious cross-tag (OXT) system from Crypto'13 [29]. The OXT scheme is quite efficient with a search complexity of $\mathcal{O}(n \cdot |\mathsf{db}(w_1)|)$, where n is the number of keywords in the conjunction and the set $\mathsf{db}(w_1)$ contains the ids of documents where the least frequent term in the conjunction w_1 appears. Static schemes like [66, 78] used pre-processing on the data to restrict this leakage.

Our focus in this chapter will be on the construction of forward and backward private dynamic SSE schemes that are adaptively secure and support conjunctive queries. We call such a scheme *conjunctive dynamic searchable symmetric encryption* (CDSSE).

The first CDSSE scheme with claimed forward and backward privacy (Type II) was proposed in 2020 [80]. This was an attempt to extend the OXT scheme to allow dynamic queries while retaining the efficiency of OXT. But, an attack in [100] showed

that the scheme is not forward private.

A few other CDSSE schemes [102, 100, 95] have followed, achieving forward privacy and different flavours of backward privacy with efficiency trade-offs. However, these schemes are far from achieving the efficiency of the OXT scheme.
In this chapter, our goal is the following.

> *Construct an adaptively secure forward and backward private CDSSE scheme that has the computational efficiency of the client and server comparable with* OXT *under standard and well-studied security definitions.*

In this work, we achieve the above goal for a class of databases, which we call as *"non-modifiable databases"*. This chapter has been reported to [3].

5.1 Non-modifiable Database

Most of the single-keyword and conjunctive dynamic SSE schemes [60, 28, 23, 25, 87, 89, 49, 101, 38, 80, 102, 100, 95] allow a user to add/delete an identifier-keyword pair (id, w) to/from the inverted index dynamically at any point of time. So we define *"modifiability"* of a document as

> *the ability to add/delete keywords to/from a document at any point.*

However, there are many real-world applications where, once the documents are uploaded to the server, they are not modified. We call such documents "*non-modifiable*". We assume the inverted index of a database storing non-modifiable documents has the following properties.

- Document identifiers (with associated keywords) can be added to the inverted index and existing identifiers (with associated keywords) can be (marked as) deleted from it *as a whole*.
- Once a document identifier (with associated keywords) has been added to the inverted index, the content of its document may change, but the associated keywords in the inverted index do not change.

We note that in this setting *modifiability can be achieved by deleting the document and then adding the updated version with a different identifier* as is common in SSE literature [28, 25, 51]. Below are some examples of applications handling non-modifiable documents.

1. *Biometric Data.* Stored biometric data (fingerprint, voice, iris scan, facial image, etc.) are typically non-modifiable. It is crucial to secure them for privacy and data protection [61], especially in large databases like Aadhaar [4, 8], for voice data collected in bulk [70], etc.

2. *Media Subscription Services.* In digital media subscription services like Netflix, Spotify, etc., the documents are the DRM-protected media files [75, 14, 5] that once uploaded, do not change.

3. *Digital Libraries and Archives*: Documents in digital libraries like the IEEE Xplore Digital Library and archives are non-modifiable and encrypted for access control [91]. They may also contain sensitive information requiring encryption [83, 82].

4. *Legal Documents*: They are non-modifiable documents containing sensitive information (personal details, financial arrangements, strategic plans, etc.) and should hence be encrypted [9]. Their keywords typically do not change.

5. *Medical Records*: Patient records are non-modifiable documents filled with personally identifiable information (PII), which must be protected under privacy laws[1], especially for access control and data sharing for collaborative research [21, 84].

6. *Financial Records*: Business and finance documents contain sensitive information about a company's financial status and strategy. Such data do not change and searchability is essential for auditing and making financial decisions. Unauthorised access or revealing access patterns can lead to financial loss, reputational damage, or legal issues.

[1]Like HIPAA in the United States: https://www.ncbi.nlm.nih.gov/books/NBK500019/

In each of these scenarios, protecting the privacy of the user "by design and by default" is a key responsibility of the service provider according to regulations like GDPR. So while the service providers may use third-party (cloud) storage services for the inverted indices, they must be encrypted to ensure user privacy. In applications like the above, modifiability is either not required (like for biometric data and digital media archives) or should be strictly prohibited (like for legal documents and medical and financial records). The existing definitions of SSE schemes and their security models do not consider this subtlety. That leaves scope for new definitions and scheme designs. As we do away with *modifiability*, several computations and storage requirements from previous SSE schemes are rendered redundant without compromising on security. Hence, we arrive at new efficient schemes that would otherwise be insecure in the general setting of SSE.

5.2 Forward and Backward Privacy in CDSSE for Non-modifiable Documents

For backward privacy, the three common notions are called BPIP/Type I, BPUP/Type II, WBP/Type III respectively. Type I is the strongest and Type III is the weakest. In [80], the security notion of the single keyword SSE (SKSSE) scheme was extended to a conjunctive SSE scheme for Type II security only. We have adopted the definition of leakage functions from [23, 25, 38, 80] and appended them as required to define leakage of CDSSE for all three standard notions of backward privacy. Also, [80] missed some leakage components in their definition. We have amended those definitions following the suggestions from [38]. In [38], authors provided another alternate definition of backward privacy called BPLP. However, in a non-modifiable database configuration, BPLP reduces to WBP. Thus we define backward privacy for BPIP, BPUP, and WBP only.

The definition of forward and backward privacy for SSE supporting conjunctive queries is similar to that of single keyword SSE schemes. In [80], the leakage of

SKSSE was extended to define the leakage for CDSSE. However, their extension is for BPUP backward privacy only. We extend the definition of forward and backward privacy of SKSSE to CDSSE for all notions of backward privacy. In [80], the definitions of $\mathsf{TimeDB}(w)$ and $\mathsf{Updates}(w)$ for a single keyword w have been extended to $\mathsf{TimeDB}(\omega)$ and $\mathsf{Updates}(\omega)$ for a conjunctive query ω. Here, we also extend the other two definitions.

Consider the query sequence $\mathsf{QS} = \{Q_1, \ldots, Q_q\}$, a conjunctive query $\omega = w_1 \wedge \cdots \wedge w_n$ and its set of included keywords $L_\omega = \{w_1, \ldots, w_n\}$. We define the following sets of leakages for ω as follows.

$\mathsf{sp}(\omega)$: The search pattern is defined for a pair of keywords w_i and w_j in the conjunction ω.

$$\mathsf{sp}(w_i, w_j) = \{u | (u, \mathsf{srch}, \omega) \in \mathsf{QS}, \text{ and } w_i, w_j \in L_\omega\}$$

Using this definition, the search pattern for the conjunction ω is defined as follows.

$$\mathsf{sp}(\omega) = \mathsf{sp}(w_1) \cup \left(\bigcup_{i=2}^{n} \mathsf{sp}(w_1, w_i) \right)$$

$\mathsf{TimeDB}(\omega)$: This is the set of all timestamped ids containing keywords in L_ω that have been added to but are yet to be deleted from the database. In [80], the definition of $\mathsf{TimeDB}(w)$ from [25] was extended to $\mathsf{TimeDB}(\omega)$. We correct the definition of $\mathsf{TimeDB}(\omega)$ from [80] with the amendment suggested in [38].

$$\mathsf{TimeDB}(\omega) = \left\{ \left(\{u_i\}_{i \in [n]}, \mathsf{id} \right) : w \in L_\omega, ((u_i, \mathsf{add}, (\mathsf{id}, w)) \in \mathsf{QS}) \wedge (\forall u > u_i, (u, \mathsf{del}, (\mathsf{id}, w)) \notin \mathsf{QS}) \right\}.$$

$\mathsf{Updates}(\omega)$: For a keyword pair (w_i, w_j) in the conjunction ω, we define the set of pairs of timestamps of update operations $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ on the database as follows.

$$\mathsf{Updates}(w_i, w_j) = \{(u, u') : ((u, \mathsf{op}, (\mathsf{id}, w_i)) \in \mathsf{QS}) \wedge ((u', \mathsf{op}, (\mathsf{id}, w_j)) \in \mathsf{QS})\}.$$

Using this definition, the update history for the conjunction ω is defined as follows.

$$\mathsf{Updates}(\omega) = \mathsf{Updates}(w_1) \cup \left(\bigcup_{i=2}^{n} \mathsf{Updates}(w_1, w_i) \right)$$

$\mathsf{DelHist}(\omega)$: We define $\mathsf{DelHist}(\omega)$ as the set of all pairs of timestamps of addition and deletion for all keywords in a given ω.

$$\mathsf{DelHist}(\omega) = \left\{ \{(u_i^{\mathsf{add}}, u_i^{\mathsf{del}})\}_{i \in [n]} : w \in L_\omega, \left((u_i^{\mathsf{add}}, \mathsf{add}, (\mathsf{id}, w)) \in \mathsf{QS} \right) \wedge \left((u_i^{\mathsf{del}}, \mathsf{del}, (\mathsf{id}, w)) \in \mathsf{QS} \right) \right\}.$$

Finally, we define the leakage function $\mathcal{L}_\Pi = (\mathcal{L}_{\Pi,\mathsf{Setup}}, \mathcal{L}_{\Pi,\mathsf{Update}}, \mathcal{L}_{\Pi,\mathsf{Search}})$, and correspondingly forward and backward privacy for a CDSSE scheme as follows.

Definition 5.2.1 (Forward Privacy of CDSSE)**.** *An $\mathcal{L}_\Pi$-adaptive secure CDSSE scheme Π is forward private if its leakages due to the* Update *protocol can be written as the following.*

$$\mathcal{L}^{\mathsf{FP}}_{\Pi,\mathsf{Update}}(\mathsf{op}, \mathsf{in}) = \mathcal{L}^{\mathsf{FP}}_{\Sigma,\mathsf{Update}}.$$

Definition 5.2.2 (Backward Privacy of CDSSE)**.** *An $\mathcal{L}_\Pi = (\mathcal{L}_{\Pi,\mathsf{Setup}}, \mathcal{L}_{\Pi,\mathsf{Search}}, \mathcal{L}_{\Pi,\mathsf{Update}})$-adaptive secure CDSSE scheme Π is backward private as per the notions* BPIP, BPUP *and* WBP *if its leakages due to the* Setup, Search *and,* Update *protocols can be written as the following.*

$$\begin{aligned}
\mathcal{L}^{\mathsf{BPIP}}_{\Pi,\mathsf{Setup}}(\mathsf{DB}) &= \emptyset, \qquad \mathcal{L}^{\mathsf{BPIP}}_{\Pi,\mathsf{Update}}(\mathsf{op}, \mathsf{id}, w) \preceq \{\mathsf{op}\}, \\
\mathcal{L}^{\mathsf{BPIP}}_{\Pi,\mathsf{Search}}(\omega) &\preceq \{\mathsf{sp}(\omega), \mathsf{TimeDB}(\omega), |\mathsf{Updates}(\omega)|\}, \\
\mathcal{L}^{\mathsf{BPUP}}_{\Pi,\mathsf{Setup}}(\mathsf{DB}) &= \emptyset, \qquad \mathcal{L}^{\mathsf{BPUP}}_{\Pi,\mathsf{Update}}(\mathsf{op}, \mathsf{id}, w) \preceq \{\mathsf{op}, w\}, \\
\mathcal{L}^{\mathsf{BPUP}}_{\Pi,\mathsf{Search}}(\omega) &\preceq \{\mathsf{sp}(\omega), \mathsf{TimeDB}(\omega), \mathsf{Updates}(\omega)\}, \\
\mathcal{L}^{\mathsf{WBP}}_{\Pi,\mathsf{Setup}}(\mathsf{DB}) &= \emptyset, \qquad \mathcal{L}^{\mathsf{WBP}}_{\Pi,\mathsf{Update}}(\mathsf{op}, \mathsf{id}, w) \preceq \{\mathsf{op}, w\}, \\
\mathcal{L}^{\mathsf{WBP}}_{\Pi,\mathsf{Search}}(\omega) &\preceq \{\mathsf{sp}(\omega), \mathsf{TimeDB}(\omega), \mathsf{Updates}(\omega), \mathsf{DelHist}(\omega)\}.
\end{aligned}$$

5.3 Our Construction

Our generic CDSSE construction uses the generic dynamic SKSSE, and the OXT abstraction of [29] to enable conjunctive search queries. The OXT protocol uses two data structures called T-Set and X-Set. The T-Set data structure is used to search all matching documents for every keyword, which is basically a single keyword SSE. The X-Set data structure stores elements that map a keyword to the document identifier which contains it. This X-Set data structure is used to search for the conjunction. The detailed description of OXT protocol is given in Figure 5-2.

In our generic CDSSE construction, we replace T-Set with any generic dynamic SKSSE for the single keyword search on the least frequent keyword w_1, and the OXT abstraction of [29] remains unaltered. The SKSSE and OXT schemes can be used unaltered, making our CDSSE completely modular. This modular nature allows the flexibility to choose efficient component modules to be used in the generic construction rather than developing a new CDSSE scheme from scratch. It creates the ground for finding better CDSSEs in the future through a more efficient SKSSE, an improved OXT, or a more suitable OXT substitute, instead of a specific construction of a CDSSE. The use of OXT also allows our CDSSE construction to achieve the search complexity of $\mathcal{O}(nu_1)$, which was previously attempted by [80].

Next, we describe our generic construction of a forward and backward private CDSSE Π for non-modifiable documents. We assume the existence of a forward and backward private DSSE Σ with support for the add operation, and the OXT scheme Γ of [29]. Π uses Σ and Γ in a modular fashion as black boxes.

Γ assumes that the client knows the *least frequent keyword* w_1 in a conjunction $\omega = w_1 \wedge \cdots \wedge w_n$. During a search operation in Γ, the server first conducts a single keyword search using the T-Set to find all ids containing the *s-term* w_1. For all such ids containing w_1, the X-Set is used to check if the *x-terms* $w_i, i \geq 2$ are also in id. In Π, Σ is used to search for the *s-term* and Γ to search for the conjunction.

A Generic DSSE Scheme. A generic DSSE scheme $\Sigma = (\Sigma.\mathsf{Setup}, \Sigma.\mathsf{Search}, \Sigma.\mathsf{Update})$ is described in Figure 5-1.

$\underline{\Sigma.\mathsf{Setup}\,(1^\lambda, \mathsf{DB})}$:
– $(\mathsf{K_s}, \sigma_\mathsf{C}, \mathsf{TSet}) \leftarrow \Sigma.\mathsf{Setup_C}(1^\lambda, \mathsf{DB})$
– $\perp \leftarrow \Sigma.\mathsf{Setup_S}(\mathsf{TSet})$
$\underline{\Sigma.\mathsf{Update}\,(k, \sigma_\mathsf{C}, \mathsf{op}, (\mathsf{id}, w); \mathsf{utk}, \mathsf{TSet})}$:
– $(\sigma_\mathsf{C}, \Sigma.\mathsf{utk}) \leftarrow \Sigma.\mathsf{Update_C}\,(\mathsf{K_s}, \sigma_\mathsf{C}, \mathsf{op}, (\mathsf{id}, w))$
– $\mathsf{TSet} \leftarrow \Sigma.\mathsf{Update_S}\,(\Sigma.\mathsf{utk}, \mathsf{TSet})$
$\underline{\Sigma.\mathsf{Search}\,(k, \sigma_\mathsf{C}, w; \mathsf{stk}, \mathsf{TSet})}$:
– $(\sigma_\mathsf{C}, \Sigma.\mathsf{stk}) \leftarrow \Sigma.\mathsf{Search_C}\,(\mathsf{K_s}, \sigma_\mathsf{C}, w)$
– $(\mathsf{V}_w, \mathsf{TSet}) \leftarrow \Sigma.\mathsf{Search_S}\,(\Sigma.\mathsf{stk}, \mathsf{TSet})$

Figure 5-1: A generic dynamic single-keyword SSE (DSSE) construction Σ

The Oblivious Cross Tag Scheme. A modular description of the OXT scheme $\Gamma = (\Gamma.\mathsf{Setup}, \Gamma.\mathsf{Add}, \Gamma.\mathsf{Search_C}, \mathsf{Search_S})$ is in Figure 5-2. It assumes the existence of three PRF families $F_p{}^{(I)} : \mathcal{K} \times \mathcal{I} \times \{0,1\} \to \mathbb{Z}_p^*$, $F_p{}^{(X)} : \mathcal{K} \times \mathcal{W} \to \mathbb{Z}_p^*$ and $F_p{}^{(Z)} : \mathcal{K} \times \mathcal{W} \times \mathbb{N} \to \mathbb{Z}_p^*$. It omits the SKSSE part of the original OXT scheme [29], to be substituted by the functionalities of the DSSE Σ. The original OXT scheme [29] only supports static databases where all (id, w) pairs are added during setup. To adapt it to our dynamic setting, we have split the setup of [29] into $\Gamma.\mathsf{Setup}$ and $\Gamma.\mathsf{Add}$ protocols, without changing their functionalities. $\Gamma.\mathsf{Search_S}$ also assumes an input V_w which has $\mathsf{V}_w[y_j] = y_j$ for $j \in [\mathsf{cnt}[w]]$ generated within Γ for a keyword w. The res however accumulates $\mathsf{V}_w[\mathsf{val}_j]$ from Σ. This step integrates Σ and Γ in a modular fashion within our CDSSE Π.

5.3.1 Our Generic CDSSE Scheme

Our generic CDSSE scheme $\Pi = (\Pi.\mathsf{Setup}, \Pi.\mathsf{Search}, \Pi.\mathsf{Update})$ is described using Σ and Γ in Figure 5-3. Note the arguments passed to $\Gamma.\mathsf{Search_S}$ in step 3 of $\Pi.\mathsf{Search_S}$ Figure 5-3. The input cnt to $\Gamma.\mathsf{Search_S}$ is $\mathsf{st.cnt}[w_1]$ from $\Pi.\mathsf{Search_S}$. The input $\Pi.\mathsf{V}_{w_1}$ is from the single keyword search $\Sigma.\mathsf{Search_S}$ on the *s-term* w_1, containing pairs (val_j, y_j) for $j \in [\mathsf{st.cnt}[w_1]]$.

On receiving the result res from the server in step 4 of $\Pi.\mathsf{Search_S}$, the client finds the search result for the conjunction. The identifier in the search result is of the form $\mathsf{id}\|\mathsf{op}$. To get the final search result $\mathsf{db}(\omega)$, the client does the following. Initialize

$\underline{\left(\mathsf{K_x}, {F_p}^{(I)}, {F_p}^{(X)}, {F_p}^{(Z)}, \mathsf{cnt}, \mathsf{XDB}\right) \leftarrow \Gamma.\mathsf{Setup}\left(1^\lambda, \mathsf{DB}\right)}$:

1. Sample $\mathsf{K_x} = (\mathsf{k_I}, \mathsf{k_X}, \mathsf{k_Z}) \in \mathcal{K}^3$, corresponding ${F_p}^{(I)}$, ${F_p}^{(X)}$ and ${F_p}^{(Z)}$,
2. Initialise an empty map $\mathsf{cnt} : \mathcal{W} \rightarrow \mathbb{Z}$
3. Set an empty set XDB.
4. Send XDB to the server.

$\underline{(\mathsf{cnt}, \Gamma.\mathsf{addtk}) \leftarrow \Gamma.\mathsf{Add}\left(\mathsf{K_x}, \mathsf{cnt}, (\mathsf{id}, w)\right)}$:

1. If $\mathsf{cnt}[w] = \perp$, then $\mathsf{cnt}[w] \leftarrow 1$; else, $\mathsf{cnt}[w] \leftarrow \mathsf{cnt}[w] + 1$
2. Compute $\mathsf{xid} \leftarrow {F_p}^{(I)}(\mathsf{k_I}, \mathsf{id})$; $x_w \leftarrow {F_p}^{(X)}(\mathsf{k_X}, w)$; $z_{\mathsf{cnt}[w]} \leftarrow {F_p}^{(Z)}(\mathsf{k_Z}, w \| \mathsf{cnt}[w])$
3. Set $y_{\mathsf{cnt}[w]} = \left(\mathsf{xid} \cdot z^{-1}_{\mathsf{cnt}[w]}\right)$; $\mathsf{xtag} \leftarrow g^{x_w \cdot \mathsf{xid}}$
4. Output $\Gamma.\mathsf{addtk} = \left(y_{\mathsf{cnt}[w]}, \mathsf{xtag}\right)$ to be sent to the server.

$\underline{(\mathsf{cnt}, \Gamma.\mathsf{xtk}) \leftarrow \Gamma.\mathsf{Search_C}\left(\mathsf{K_x}, \mathsf{cnt}, \omega\right)}$:

1. For $(j = 1, \ldots, \mathsf{cnt}[w_1])$
   For $(i = 2, \ldots, n)$
   $\mathsf{xtk}[j, i] \leftarrow g^{{F_p}^{(Z)}(\mathsf{k_Z}, w_1 \| j) \cdot {F_p}^{(X)}(\mathsf{k_X}, w_i)}$
   End For
   Randomly shuffle $\mathsf{xtk}[j, i]$.
   End For
2. Output $\Gamma.\mathsf{xtk} = \{\mathsf{xtk}[j, 2], \ldots, \mathsf{xtk}[j, n]\}$ to be sent to the server.

$\underline{\mathsf{res} \leftarrow \Gamma.\mathsf{Search_S}\left(\mathsf{cnt}, \Gamma.\mathsf{xtk}, \mathsf{V}_w, \mathsf{XDB}\right)}$:

1. $\mathsf{res} \leftarrow \emptyset$.
2. For $(j = 1, \ldots, \mathsf{cnt})$
   A. $\mathsf{flag} \leftarrow \mathsf{true}$
   B. For $(i = 2, \ldots, n)$
      i. If $(\Pi.\mathsf{xtk}[j, i]^{\mathsf{V}_w[y_j]} \notin \mathsf{XDB})$,
         a. $\mathsf{flag} \leftarrow \mathsf{false}$
   C. If $(\mathsf{flag} == \mathsf{true})$,
      i. $\mathsf{res} \leftarrow \mathsf{res} \cup \{\mathsf{V}_w[\mathsf{val}_j]\}$
3. Return res

Figure 5-2: A modular description of Γ without the SKSSE part of [29]

an empty set $\mathcal{I}_\omega \leftarrow \emptyset$. For each $r \in \mathsf{res}$, parse $r = \mathsf{id} \| \mathsf{op}$. Now if $(\mathsf{op} = \mathsf{add})$, $\mathcal{I}_\omega \leftarrow \mathcal{I}_\omega \cup \{\mathsf{id}\}$, and if $(\mathsf{op} = \mathsf{del})$. $\mathcal{I}_\omega \leftarrow \mathcal{I}_\omega \setminus \{\mathsf{id}\}$. Finally output, $\mathcal{I}_\omega$.

5.3.2 Design Rational

With our generalisation in the previous section, the scheme of [80] is now just an instantiation of our construction. The work of [80] essentially club Mitra [49] and their proposed ODXT. The scheme of [80] has a large search token size due to the

$\Pi.\mathsf{Setup}\left(1^\lambda, \mathsf{DB}\right)$:

Client:

$(\mathsf{K}, \mathsf{st}, \Pi.\mathsf{EDB}) \leftarrow \Pi.\mathsf{Setup}_\mathsf{C}\left(1^\lambda, \mathsf{DB}\right)$:

$(\mathsf{K_s}, \sigma_\mathsf{C}, \mathsf{TSet}) \leftarrow \Sigma.\mathsf{Setup}_\mathsf{C}(\mathsf{DB}, 1^\lambda)$; assign $\mathsf{st}.\sigma_\mathsf{C} \leftarrow \sigma_\mathsf{C}$ and $\Pi.\mathsf{TSet} \leftarrow \mathsf{TSet}$

$(\mathsf{K_x}, {F_p}^{(I)}, {F_p}^{(X)}, {F_p}^{(Z)}, \mathsf{cnt}, \mathsf{XDB}) \leftarrow \Gamma.\mathsf{Setup}(\mathsf{DB}, 1^\lambda)$

Assign $\mathsf{st.cnt} \leftarrow \mathsf{cnt}$ and $\Pi.\mathsf{XDB} \leftarrow \mathsf{XDB}$

Set $\mathsf{K} \leftarrow (\mathsf{K_s}, \mathsf{K_x})$, and $\mathsf{st} \leftarrow (\mathsf{st}.\sigma_\mathsf{C}, \mathsf{st.cnt})$

Send, $\Pi.\mathsf{EDB} \leftarrow (\Pi.\mathsf{TSet}, \Pi.\mathsf{XDB})$ to the server

Server:

$\perp \leftarrow \Pi.\mathsf{Setup}_\mathsf{S}\left(\Pi.\mathsf{EDB}\right)$: Server stores $\Pi.\mathsf{EDB}$

$\Pi.\mathsf{Update}\left(k, \mathsf{st}, \mathsf{op}, (\mathsf{id}, w); \Pi.\mathsf{utk}, \Pi.\mathsf{EDB}\right)$:

Client:

$(\mathsf{st}, \Pi.\mathsf{utk}) \leftarrow \Pi.\mathsf{Update}_\mathsf{C}\left(\mathsf{K}, \mathsf{st}, \mathsf{op}, (\mathsf{id}, w)\right)$:

1. Call $\Sigma.\mathsf{Update}_\mathsf{C}\left(\mathsf{K_s}, \mathsf{st}.\sigma_\mathsf{C}, \mathsf{add}, (w, (\mathsf{id}\|\mathsf{op}))\right)$ to get $\Sigma.\mathsf{utk}$
2. Call $\Gamma.\mathsf{add}\left(\mathsf{K_x}, \mathsf{st.cnt}, w, \mathsf{id}\|\mathsf{op}\right)$ to get $\Gamma.\mathsf{addtk} = \left(y_{\mathsf{cnt}[w]}, \mathsf{xtag}\right)$
3. Set $\Pi.\mathsf{utk}_s \leftarrow (\Sigma.\mathsf{utk}, y_{\mathsf{cnt}[w]})$ and $\Pi.\mathsf{utk}_x \leftarrow \{\mathsf{xtag}\}$;
4. Send $\Pi.\mathsf{utk} \leftarrow (\Pi.\mathsf{utk}_s, \Pi.\mathsf{utk}_x)$ to the server

Server:

$\Pi.\mathsf{EDB} \leftarrow \Pi.\mathsf{Update}_\mathsf{S}\left(\Pi.\mathsf{utk}, \Pi.\mathsf{EDB}\right)$:

1. Call $\Sigma.\mathsf{Update}_\mathsf{S}(\Pi.\mathsf{utk}_s, \Pi.\mathsf{TSet})$ and to get the output TSet as following
   a. Parse $(\Sigma.\mathsf{utk}, y_{\mathsf{cnt}[w]}) \leftarrow \Pi.\mathsf{utk}_s$
   b. Further parse $(\mathsf{loc}, \mathsf{val}) \leftarrow \Sigma.\mathsf{utk}$
   c. Update $\Pi.\mathsf{TSet}[\mathsf{loc}] \leftarrow (\mathsf{val}, y_{\mathsf{cnt}[w]})$
2. Set $\Pi.\mathsf{XDB}$ as $\Pi.\mathsf{XDB} \leftarrow \Pi.\mathsf{XDB} \cup \Pi.\mathsf{utk}_x$
3. Output $\Pi.\mathsf{EDB} \leftarrow (\Pi.\mathsf{TSet}, \Pi.\mathsf{XDB})$

$\Pi.\mathsf{Search}\left(k, \mathsf{st}, \omega; \Pi.\mathsf{stk}, \Pi.\mathsf{EDB}\right)$:

Client:

$(\Pi.\mathsf{stk}, \Pi.\mathsf{xtk}) \leftarrow \Pi.\mathsf{Search}_\mathsf{C}\left(\mathsf{K}, \mathsf{st}, \omega\right)$:

1. Call $\Sigma.\mathsf{Search}_\mathsf{C}\left(\mathsf{K_s}, \mathsf{st}.\sigma_\mathsf{C}, w_1\right)$ to get $\Sigma.\mathsf{stk}$; assign $\Pi.\mathsf{stk} \leftarrow \Sigma.\mathsf{stk}$
2. Call $\Gamma.\mathsf{Search}\left(\mathsf{K_x}, \mathsf{st.cnt}, \omega\right)$ to get $\Gamma.\mathsf{xtk}$; assign $\Pi.\mathsf{xtk} \leftarrow \Gamma.\mathsf{xtk}$
3. Send $(\Pi.\mathsf{stk}, \Pi.\mathsf{xtk})$ to the server

Server:

$\mathsf{res} \leftarrow \Pi.\mathsf{Search}_\mathsf{S}\left((\Pi.\mathsf{stk}, \Pi.\mathsf{xtk}), \Pi.\mathsf{EDB}\right)$:

1. $\mathsf{res} \leftarrow \phi$.
2. Call $\Sigma.\mathsf{Search}_\mathsf{S}\left(\Pi.\mathsf{stk}, \Pi.\mathsf{TSet}\right)$ to get $\Pi.\mathsf{V}_{w_1} = \{(\mathsf{val}_j, y_j) : j \in [\mathsf{st.cnt}[w_1]]\}$
3. $\mathsf{res} \leftarrow \Gamma.\mathsf{Search}_\mathsf{S}\left(\mathsf{st.cnt}, \Pi.\mathsf{xtk}, \Pi.\mathsf{V}_{w_1}, \Pi.\mathsf{XDB}\right)$
4. Send res to the client

Figure 5-3: Our generic CDSSE construction Π.

use of Mitra. With our generalisation, we can do away with the large search token size of [80] via our instantiations Π_{BP}-OXT which performances better as can be seen

from our experimental results.

Using a generic SKSSE and the OXT scheme in a black box fashion we achieved the computation and communication complexity for search operation as $\mathcal{O}(n \cdot \mathsf{db}(w_1))$, and update operation of $\mathcal{O}(1)$ for our construction. In [95], the computation and communication complexities of the search operation are both $\mathcal{O}(u_1 + n \cdot \mathsf{db}(w_1))$ and involves two rounds, where u_1 is the number of updates related to w_1; the complexities of the update operation are $\mathcal{O}(|W|/|W_d|)$ for adding a document, $\mathcal{O}(|D|)$ for editing and $\mathcal{O}(1)$ for deleting, where $|W|$ is the number of keywords in the database and $|W_d|$ denotes the number of keywords contained in the updated document. The storage requirement of [95] is $\mathcal{O}(|W||D|)$, as compared to $\mathcal{O}(N)$ for our scheme, where N is the number of identifier-keyword pairs in the database. Typically, N is much smaller than $|W||D|$ in practice. There is no additional storage requirement due to the keyword updates in [95]. Compared to these, our scheme enjoys the search and update complexities as $\mathcal{O}(n \cdot \mathsf{db}(w_1))$, and $\mathcal{O}(1)$ respectively.

The scheme of [102] performs $\mathcal{O}(|D|)$ computation for search, where $|D|$ is the number of documents in the database, making it prohibitively slow. The scheme of [100] uses a single keyword SSE to store and fetch the elements of both TSet and XSet which makes it very slow as well. The scheme of [95] runs in sub-linear time but is still far from the practical performance of OXT. See Table 5.1 for a detailed comparison.

Scheme	Search			Update			Storage		FP	BP	KPRP
	Computation	Communication	RT	Add	Edit	Delete	Client	Server			
VBTree [93]	$\mathcal{O}(u\lvert\mathsf{db}(w_1)\rvert \log \lvert D\rvert)$	$\mathcal{O}(n+\lvert\mathsf{db}(\omega)\rvert)$	1	$\mathcal{O}(W_d)$	$\mathcal{O}(v)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert W\rvert \log \lvert D\rvert)$	$\mathcal{O}(\lvert W\rvert h)$	Yes	No	No
BDXT [80]	$\mathcal{O}(u_1+n\lvert\mathsf{db}(w_1)\rvert)$	$\mathcal{O}(u_1+n\lvert\mathsf{db}(w_1)\rvert)$	2	$\mathcal{O}(1)$	-	$\mathcal{O}(1)$	$\mathcal{O}(\lvert W\rvert \log \lvert D\rvert)$	$\mathcal{O}(N)$	No	Type-II	No
ODXT [80]	$\mathcal{O}(nu_1)$	$\mathcal{O}(nu_1)$	1	$\mathcal{O}(1)$	-	$\mathcal{O}(1)$	$\mathcal{O}(\lvert W\rvert \log \lvert D\rvert)$	$\mathcal{O}(N)$	No	Type-II	No
FBSSE-CQ [102]	$\mathcal{O}(u\lvert D\rvert)$	$\mathcal{O}(n+\lvert D\rvert)$	1	$\mathcal{O}(\lvert D\rvert)$	$\mathcal{O}(\lvert D\rvert)$	$\mathcal{O}(\lvert D\rvert)$	$\mathcal{O}(\lvert W\rvert \log \lvert D\rvert+\lambda)$	$\mathcal{O}(N\lvert D\rvert)$	Yes	Type-II	Yes
HDXT [95]	$\mathcal{O}(u_1+n\lvert\mathsf{db}(w_1)\rvert)$	$\mathcal{O}(u_1+n\lvert\mathsf{db}(w_1)\rvert)$	2	$\mathcal{O}(\lvert W\rvert/W_d)$	$\mathcal{O}(\lvert D\rvert)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert W\rvert \log \lvert D\rvert+\lambda)$	$\mathcal{O}(\lvert W\rvert\lvert D\rvert)$	Yes	Type-II	Yes
Our construction (Sec 5.3)	$\mathcal{O}(nu_1)$	$\mathcal{O}(nu_1)$	1	$\mathcal{O}(1)$	$\mathcal{O}(W_d)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert W\rvert \log \lvert D\rvert)$	$\mathcal{O}(N)$	NMD	Type-II	No

Table 5.1: Comparison of our construction with previous schemes. W: the set of keywords. D: Set of documents. W_d: Number of keywords in the document to be updated. N: Number of identifier-keyword pairs in the database. n: Number of keywords in a conjunction. u_i: Updates performed for keyword w_i. $u = \sum_{i=1}^{n} u_i$: Total updates for a conjunction of n keywords. v: Keywords involved in the edit operation. h: Average number of documents matched by any two keywords. $\mathsf{db}(\omega)$: Set of matching document identifiers for query $\omega = w_1 \wedge \cdots \wedge w_n$, where $n \geq 1$. RT: Round-Trip. FP: Forward Private. BP: Backward Private. KPRP: Keyword-Pair Result Pattern Hiding. NMD: Non-Modifiable Documents.

Leakage Suppression. The scheme of [95] pays with extra storage, computation and an additional round-trip to prevent *keyword-pair result pattern* (KPRP) leakage. In a conjunctive query for $w_1 \wedge w_2 \wedge \cdots \wedge w_n$, KPRP leakage contains $\mathsf{db}(w_i) \cap \mathsf{db}(w_j)$ for all $1 \leq i < j \leq n$. Although our scheme does not prevent KPRP leakage completely, we have taken some countermeasures prescribed in [80] to restrict the leakage of our scheme that makes the attack of [97] using KPRP leakage very difficult to mount. We have discussed these countermeasures later while discussing the security of our scheme. According to [80] using those countermeasures, the best-known attack described in [97] which uses KPRP leakage of SSE, has a success probability of less than 5% even when the fraction of file injection is more than 60% in the database. By the same argument, despite not hiding the KPRP leakage, our construction is suitable for practical applications.

5.4 Security of Our Construction

The security of SSE schemes is generally argued based on leakage to the server that is upper bounded by some well-defined and "*sensible*" leakage functions that have been studied over many years. The practical consequences of such leakage functions especially through resulting attacks, are well studied and understood. So it is always good to have an SSE scheme that is secure against a well-studied leakage function. Our generic construction enjoys this advantage, as the security of our scheme is inherited from the security of the individual components that are secure under well-studied leakage functions. We note here that previous forward and backward private CDSSEs [102] and [100] allowing "modifiable" documents have proven their security under non-standard security notions that warrant more investigations.

We provide a rigorous security analysis of our generic scheme, assuming the standard and maximum possible leakage of the individual components. The reason to do so is to keep room for possible future improvements in efficiency or the adaptability of the generic construction in other ways.

We recollect that Σ is an adaptively secure forward and backward private SKSSE

scheme as per Definitions 2.5.1, 2.5.5 and 2.5.6. With this assumption, we prove our generic CDSSE scheme Π to be adaptively secure as well. The leakage functions in the backward privacy definition of CDSSE (see Definition 5.2.1 and Definition 5.2.2) subsume the actual leakages of our generic scheme. As discussed in Section 5.2, the leakage of WBP is a superset of the leakage of the other two notions of backward privacy – BPUP and BPIP – for non-modifiable documents, and that of forward privacy as well. We prove Π to be secure assuming the leakage for WBP. So our scheme would be secure when instantiated with a BPIP and BPUP secure SKSSE scheme as well. We follow the common practice of defining and subsequently proving the security of SSE, assuming more than the actual leakages. We later provide the exact leakages of our scheme as well. We assume the leakage of our conjunctive dynamic SSE Π to be the following.

$$\mathcal{L}_{\Pi,\mathsf{Update}}(\mathsf{op}, \mathsf{id}, w) = \mathsf{op}, \qquad \mathcal{L}_{\Pi,\mathsf{Search}}(\omega) = \{\mathsf{sp}(\omega), \mathsf{TimeDB}(\omega), \mathsf{Updates}(\omega)\}\,.$$

Note that our protocol never used del as an operation, so $\mathsf{DelHist}(\omega)$ is always empty, in our case. To the best of our knowledge, all forward and backward private schemes known in the literature have, at most, this leakage. Also, it is essential to point out that the leakage of our construction becomes exactly $\mathcal{L}_\Sigma$ if $\omega = w$.

Theorem 5.4.1. *Assuming that (1) Σ is an $\mathcal{L}_\Sigma$-adaptively secure dynamic DSSE, (2) F_p is a secure PRF, and (3) the eddh problem is hard in the group output by $\mathcal{G}$, our scheme Π is $\mathcal{L}_\Pi^{\mathsf{BPUP}}$-adaptively secure conjunctive dynamic SSE.*

Proof Idea. We first describe six games $\mathsf{G}_0, \ldots, \mathsf{G}_6$ where we replace all the cryptographic primitives used in our construction one by one with ideal primitives and show that no PPT adversary can detect the change in any of the steps. Finally, we show that there exists a simulator that takes the leakages $\mathcal{L}_\Pi = (\mathcal{L}_{\Pi,\mathsf{Setup}}, \mathcal{L}_{\Pi,\mathsf{Update}}, \mathcal{L}_{\Pi,\mathsf{Search}})$ as input and simulates the protocol's transcript that is indistinguishable from the actual execution of the protocol with ideal primitives. If all leakages have been considered, then the simulator can simulate all the server's inputs. Since the server has no other

source of input in the protocol, hence there is no other leakage. This proves that our construction Π does not leak more than what is defined by the leakage $\mathcal{L}_\Pi$.

In our security proof, Game G_0 is the original execution of the protocol Π. In Game G_1 replace all cryptographic primitives used in the Σ protocol with ideal primitives, and show that the adversary has an advantage at most as an SKSSE adversary against the secure Σ protocol. In Game G_2 we replace all the PRFs $F_p^{(I)}$, $F_p^{(X)}$, and $F_p^{(Z)}$ one by one with randomly sampled elements from $\mathbb{Z}_p^*$, and show that an adversary has negligible advantage in Game G_2 over Game G_1. In Game G_3, we simulate the part of the search token of Π in the same fashion as proof of the underlying Σ and show that an adversary has a negligible advantage in Game G_2 over Game G_3, which is upper bounded by the SKSSE adversary. In Game G_4 and Game G_5 we change the way xtk and $y_{\mathsf{cnt}[w]}$ are generated respectively and show Game G_3, G_4 and Game G_5 are all equivalent. In the final Game G_6 we change the way xtags are generated and show that for all adversary the distinguishing advantage of G_6 over G_5 is upper bounded by the eddh advantage of any PPT adversary. Finally, we demonstrate a poly-time simulator that can simulate a computationally indistinguishable transcript from what the adversary got in Game G_6. Thus, we conclude our theorem.

Proof. We now describe the six games $\mathsf{G}_0, \mathsf{G}_1, \ldots, \mathsf{G}_6$ in the following.

Game G_0. The experiment starts with the real execution of the protocol Π as described in Section 5.3.1. Thus we have,

$$\Pr\left[\mathsf{SSEReal}_{\mathcal{A}}^{\Pi}(1^\lambda, q) = 1\right] = \Pr[\mathsf{G}_0 = 1].$$

As we have assumed Σ to be adaptively secure, so it can be simulated exactly as in its security proof. So in the following steps, we do not explicitly argue about the simulation of those elements generated from Σ.

Game G_1. In game G_1, we replace all cryptographic primitives used in the construction of $\Sigma.\mathsf{utk}$, same as the done in the security proof of Σ. We can do so as Σ is $\mathcal{L}_\Sigma$-adaptive secure. Thus the advantage of any adversary distinguishing game G_0

and G_1 is bounded by the SSE advantage of Σ. We write it as follows.

$$|\Pr[\mathsf{G}_0 = 1] - \Pr[\mathsf{G}_1 = 1]| \leq \mathsf{Adv}^{\Sigma,\mathsf{SSE}}_{\mathcal{A},\mathcal{S},\mathcal{L}_\Sigma} \leq \mathsf{negl}(\lambda).$$

GAME G_2. Next the challenger replaces all executions of ${F_p}^{(I)}$, ${F_p}^{(X)}$ and ${F_p}^{(Z)}$ one by one with randomly sampled strings from $\mathbb{Z}^*_p$. The experiment maintains three lists I, X and Z to store the values sampled for ${F_p}^{(I)}$, ${F_p}^{(X)}$ and ${F_p}^{(Z)}$, thus can be reused later if needed. The entries of I, X and Z are indexed by $(\mathsf{id}, \mathsf{op})$[2], w and $(w, \mathsf{cnt}[w])$. For an update operation of $(\mathsf{op}, (\mathsf{id}, w))$, if $I[\mathsf{op}, \mathsf{id}] = \perp$ then the challenger picks a random value form $\mathbb{Z}^*_p$ and uses it for ${F_p}^{(I)}(\mathsf{k_I}, \mathsf{id}\|\mathsf{op})$, and also stores the random value in $I[\mathsf{op}, \mathsf{id}]$. If $I[\mathsf{op}, \mathsf{id}] \neq \perp$, then the challenger uses the value stored in $I[\mathsf{op}, \mathsf{id}]$. The other two PRFs ${F_p}^{(X)}$ and ${F_p}^{(Z)}$ are simulated similarly. Now if there exist PPT adversaries $\mathcal{A}_I$, $\mathcal{A}_X$ and $\mathcal{A}_Z$ who can distinguish between the games G_1 and G_2 based on the use of ${F_p}^{(I)}$, ${F_p}^{(X)}$ and ${F_p}^{(Z)}$ respectively, we can use these adversaries to break the security of F_p. Thus,

$$\begin{aligned}|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_2 = 1]| &\leq \mathsf{Adv}^{\mathsf{prf}}_{{F_p}^{(I)},\mathcal{A}_I}(1^\lambda) + \mathsf{Adv}^{\mathsf{prf}}_{{F_p}^{(X)},\mathcal{A}_X}(1^\lambda) + \mathsf{Adv}^{\mathsf{prf}}_{{F_p}^{(Z)},\mathcal{A}_Z}(1^\lambda) \\ &\leq \mathsf{negl}(\lambda).\end{aligned}$$

GAME G_3. For a search query $\omega = w_1 \wedge w_2 \wedge \cdots \wedge w_n$ in step 1 of $\Pi.\mathsf{Search}_\mathcal{C}$ the client generates $\Pi.\mathsf{stk}$ using $\Sigma.\mathsf{stk}$. In the game G_3, the challenger proceeds exactly as the security proof of Σ to show the generation of $\Sigma.\mathsf{stk}$ is indistinguishable from its simulated form. The same argument will hold in this proof as well. Since the difference between G_2 and G_3 is only in the generation of $\Sigma.\mathsf{stk}$ (using ideal primitives) and the advantage of any such adversary is at most $\mathsf{Adv}^{\mathsf{sse}}_{\Sigma,\mathcal{A},\mathcal{S}}$, hence we have

$$|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_3 = 1]| \leq \mathsf{Adv}^{\Sigma,\mathsf{SSE}}_{\mathcal{A},\mathcal{S},\mathcal{L}_\Sigma} \leq \mathsf{negl}(\lambda).$$

GAME G_4. For a search query $\omega = w_1 \wedge w_2 \wedge \cdots \wedge w_n$, the challenger next changes the

[2]Note that the input to ${F_p}^{(I)}$ is id along with the key $\mathsf{k_I}$. However, when called, $\mathsf{id}||\mathsf{op}$ is passed in place of id and hence the simulator will index the list by $(\mathsf{id}, \mathsf{op})$.

manner in which $\mathsf{xtk}_{i,j}$ is generated. Recall, $\mathsf{xtk}_{i,j} = g^{F_p{}^{(Z)}(\mathsf{k_Z}, w_1 \| j) \cdot F_p{}^{(X)}(\mathsf{k_X}, w_i)}$, where $j \in [\mathsf{cnt}[w]]$ and $i \in [2, n]$. The challenger knows all ids that were updated corresponding to the *s-term* w_1. The challenger computes the xtag (as computed in steps 2 and 3 of $\Pi.\mathsf{Update}_{\mathcal{C}}$) values corresponding to all the ids that were updated for w_1 and all the keywords in the *x-term*s. Notice that the challenger can do this using its lists I and X. Then the challenger computes the values for $y_{i,j} = F_p{}^{(I)}(\mathsf{k_I}, \mathsf{id} \| \mathsf{op}) \cdot F_p{}^{(Z)}(\mathsf{k_Z}, w \| \mathsf{cnt}[w])$ for all the ids updated for w_1 and for all keywords in the *x-term*s using the lists I and Z. Finally the challenger computes $\mathsf{xtk}_{i,j} = \mathsf{xtag}_{i,j}^{y_{i,j}^{-1}}$. Now, the view of the adversary in games G_3 and G_4 are identical, and no adversary has any advantage of distinguishing between these two games. Thus,

$$\Pr[\mathsf{G}_3 = 1] = \Pr[\mathsf{G}_4 = 1].$$

Game G_5. Next, the challenger samples random values $y \xleftarrow{\$} \mathbb{Z}_p^*$ and uses them in place of $y_{\mathsf{cnt}[w]}$'s in the update operations. We recollect that $y_{\mathsf{cnt}[w]} = \mathsf{xid} \cdot z_{\mathsf{cnt}[w]}^{-1} \in \mathbb{Z}_p^*$. Now if $a, b, c \xleftarrow{\$} \mathbb{Z}_p^*$, then c is statistically indistinguishable from $(a \cdot b) \in \mathbb{Z}_p^*$ for any adversary. The secure PRF F_p is used to generate xid and $z_{\mathsf{cnt}[w]}$. So xtag and $z_{\mathsf{cnt}[w]}^{-1}$ are also indistinguishable for any random element from $\mathbb{Z}_p^*$. In the Games G_2, we have already replaced all uses of F_p with randomly sampled values from $\mathbb{Z}_p^*$. Replacing the value of $y_{\mathsf{cnt}[w]}$ with a randomly sampled element from $\mathbb{Z}_p^*$ does not change the advantage of any adversary. Thus,

$$\Pr[\mathsf{G}_4 = 1] = \Pr[\mathsf{G}_5 = 1].$$

Game G_6. In game G_6, for each update operation, the challenger samples a random value $\gamma \xleftarrow{\$} \mathbb{Z}_p^*$ and replaces the value of xtag in every update operation with the value g^{γ}. Recall that $\mathsf{xtag} = g^{F_p{}^{(I)}(\mathsf{k_I}, \mathsf{id}) \cdot F_p{}^{(X)}(\mathsf{k_X}, w)}$ where F_p is a secure PRF. We have already replaced the values of every execution of F_p with random values from $\mathbb{Z}_p^*$. In particular, we already have $\mathsf{xtag} = g^{\alpha \cdot \beta}$ for random values $\alpha, \beta \in \mathbb{Z}_p^*$. Upon replacing the values of xtag with g^{γ} if any adversary $\mathcal{A}$ can detect the change between the games G_5 and G_6, we can construct another adversary $\mathcal{B}$ which uses $\mathcal{A}$

as a subroutine to break the eddh problem. Due to [80], breaking the eddh problem is equivalent to breaking the ddh assumption if the number of queries is bounded by $q = \mathsf{poly}(\lambda)$. Thus,

$$|\Pr[\mathsf{G}_5 = 1] - \Pr[\mathsf{G}_6 = 1]| \leq \mathsf{Adv}^{\mathsf{eddh}}_{\mathcal{B}} \leq \mathsf{negl}(\lambda).$$

Simulation. Finally, we demonstrate a simulator that perfectly simulates a transcript, which would be indistinguishable from the transcript in game G_6. The simulator does not have the actual queries in Π. Instead, the simulator only has access to the leakage functions $\mathcal{L}_\Pi$. The simulation has two parts, one is to simulate the components generated in the execution of Σ, and another is to simulate the components of Γ, which together construct Π. Now the components generated using Σ can be simulated perfectly as Σ is $\mathcal{L}_\Sigma$-adaptively secure SSE. We note that $\mathcal{L}_\Sigma \subseteq \mathcal{L}_\Pi$.

We demonstrate how to simulate the components generated in the execution of Γ. To simulate update queries, the simulator has access to leakage function $\mathcal{L}^{\mathsf{BPUP}}_{\Pi,\mathsf{Update}}(\mathsf{op}, w, \mathsf{id}) \preceq \{\mathsf{op}, w\}$. The simulator knows all the time stamps of update queries. For every update query, the simulator simulates $y_{\mathsf{cnt}[w]}$ and xtag by sampling two uniform random values α, β from $\mathbb{Z}^*_p$ and provides α and g^β in place of $y_{\mathsf{cnt}[w]}$ and xtag. This is the same as what is done in game G_6. So the simulation is indistinguishable. The simulator also stores the values of $y_{\mathsf{cnt}[w]}$ and xtag sampled for each update in a map Map indexed by the timestamps of the queries.

Now to simulate a conjunctive search query $\omega = w_1 \wedge w_2 \wedge \cdots \wedge w_n$, the simulator simulates $\mathsf{xtk}_{i,j}$ as follows. From the leakage functions $\mathsf{Updates}(w)$ and $\mathsf{TimeDB}(w)$, the simulator knows all timestamps of update queries for the *s-term* and their corresponding ids. From $\mathsf{TimeDB}(\omega)$ and $\mathsf{Updates}(\omega)$, the simulator knows the timestamps where a keyword in the *x-term* matches these ids. The simulator uses these timestamps to determine the corresponding $\mathsf{xtag}_{i,j}$ and $y_{i,j}$ values to construct

$$\mathsf{xtk}_{i,j} = \mathsf{xtag}_{i,j}^{y^{-1}_{\mathsf{cnt}_{i,j}}}.$$

This step is similar to what is done in game G_6. So the simulation is indistinguishable.

Lastly, if two search queries ω_1 and ω_2 have different *s-terms* but the same *x-term* and share some common ids, then the simulator should simulate the xtk such that the same xtag is generated for those queries. The simulator knows this by considering $\mathsf{sp}(\omega_1)$, $\mathsf{Updates}(\omega_1)$, $\mathsf{sp}(\omega_2)$ and $\mathsf{Updates}(\omega_2)$. Hence this simulation is also indistinguishable, and we have

$$\Pr\left[\mathsf{SSEIdeal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}_\Pi}(1^\lambda, q) = 1\right] = \Pr[\mathsf{G}_6 = 1].$$

Thus,

$$\left|\Pr\left[\mathsf{SSEReal}^{\Pi}_{\mathcal{A}}(1^\lambda, q) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}_\Pi}(1^\lambda, q) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

□

5.5 Leakage Analysis

Although we have argued the security of our scheme assuming the standard and maximum possible leakage functions defined in the literature, we also argue about the precise leakage of our scheme. It has been found historically that without a proper leakage analysis SSE has been attacked by exploiting those leakages missed in the analysis. Like leakage miss in dynamic SSE was exploited in [97]. Also, the update leakage of [80] was assumed to be empty. Whereas, in some cases, it leaks the keyword, as shown in [102]. Thus, a thorough leakage analysis is essential for any SSE. In order to achieve this, we analyze the leakage of our scheme through all possible combinations of queries. This novel technique gives confidence in the actual leakage of any SSE scheme. This discussion helps in understanding the consequences of the effect of combining different secure components and the exact leakage of the combination.

The actual leakage of our CDSSE construction is much less than those assumed above in Section 5.4. The following are public at the beginning of Π: the security parameter λ, the description of Π, the set of all possible keywords $\mathcal{W}$, the set db from

which each document is assigned a unique identifier, the key space $\mathcal{K}$, the descriptions of the data-structures $\Pi.\mathsf{TSet}$ and $\Pi.\mathsf{XDB}$ to be stored in the server, the descriptions of the PRF families $F_p^{(I)}(\cdot,\cdot)$, $F_p^{(X)}(\cdot,\cdot)$, $F_p^{(Z)}(\cdot,\cdot)$.

Setup Leakage

In the setup phase, we assume that the initial database DB that is input to $\Sigma.\mathsf{Setup}_\mathsf{C}$ is empty. If the initial database is not empty, we would outsource an empty database in the setup phase and then update the database using the update protocol. Hence, the initial structures of $\Pi.\mathsf{TSet}$ and $\Pi.\mathsf{XDB}$ are already known to the adversary. The server receives as input $\Pi.\mathsf{EDB} = (\Pi.\mathsf{TSet}, \Pi.\mathsf{XDB})$ and stores it. Since $\Pi.\mathsf{EDB}$ does not have any data from db or $\mathcal{W}$, there is no leakage during the setup of Π. In other words, $\mathcal{L}_{\Pi,\mathsf{Setup}} = \emptyset$.

After the setup, the server receives additional information with each update or search query. To find the leakage due to a query $q_i \in \mathsf{QS}$, we should consider all information that the server had before q_i and those that it attains due to and after processing q_i.

Update Leakage

Single Update Leakage. Before an update query for $(\mathsf{op}, (\mathsf{id}, w))$, let the state of the client be $\mathsf{st} = (\mathsf{st}.\sigma_\mathsf{C}, \mathsf{st}.\mathsf{cnt})$ while the server has the encrypted database $\Pi.\mathsf{EDB} = (\Pi.\mathsf{TSet}, \Pi.\mathsf{XDB})$. During an update operation, the client calls $\Pi.\mathsf{Update}_\mathsf{C}$ and the output $\Pi.\mathsf{utk} = \left(\left(\Sigma.\mathsf{utk}, y_{\mathsf{cnt}[w]}\right), \mathsf{xtag}\right)$ is sent to the server. So the leakage due to the update operation must be from $\Pi.\mathsf{utk}$ only. Note that $\Sigma.\mathsf{utk}$ is an output of $\Sigma.\mathsf{Update}_\mathsf{C}$ and $(y_{\mathsf{cnt}[w]}, \mathsf{xtag})$ is an output of $\Gamma.\mathsf{Add}$. We argue that these two outputs together do not leak any more than $\mathcal{L}_\Pi$.

1. $\Sigma.\mathsf{utk}$ *at most leaks* $\emptyset$. We first note that the protocol Π calls $\Sigma.\mathsf{Update}_\mathsf{C}$ only with $\mathsf{op} = \mathsf{add}$ and never uses $\mathsf{op} = \mathsf{del}$. By Definition 2.5.5 and Definition 2.5.6 of forward and backward privacy of Σ respectively, the update token $\Sigma.\mathsf{utk}$ output by $\Sigma.\mathsf{Update}_\mathsf{C}(\mathsf{K}_\mathsf{s}, \sigma_\mathsf{C}, \mathsf{op}, (\mathsf{id}, w))$ leaks at most the operation op which is

always add in Π. However, as for both cases $\mathsf{op} = \mathsf{add}$ as well as $\mathsf{op} = \mathsf{del}$ the algorithm only issues an add token, so add and del operations are indistinguishable. Hence, there is no leakage from $\Sigma.\mathsf{utk}$.

2. xtag *at most leaks* $\emptyset$. Recall that, $x_w = F_p^{(X)}(\mathsf{k_X}, w)$ is fixed for every w, and $\mathsf{xid} = F_p^{(I)}(\mathsf{k_I}, \mathsf{id}\|\mathsf{op})$ is fixed for every $(\mathsf{id}\|\mathsf{op})$. In our setting for non-modifiable documents, the tuple $(\mathsf{op}, w, \mathsf{id})$ is always unique for every update operation (as (id, w) is added or deleted only once). The pseudo-randomness of F_p ensures that x_w and xid for a single update are two random elements from the group G. Thus ddh hardness of G (Definition 2.2.2) ensures that no efficient adversary can distinguish the xtag which is $g^{x_w \cdot \mathsf{xid}}$ from a random element of G.

3. $y_{\mathsf{cnt}[w]}$ *at most leaks* $\emptyset$. The value $\mathsf{cnt}[w]$ changes (incremented by 1) for every update operation corresponding to every keyword w and never repeats. As a result, $(w\|\mathsf{cnt}[w])$ is always unique. The pseudo-randomness of F_p ensures that $z_{\mathsf{cnt}[w]}$ which is $F_p{}^{(Z)}(\mathsf{k_Z}, w\|\mathsf{cnt}[w])$ and consequently $y_{\mathsf{cnt}[w]}$ which is $\mathsf{xid} \cdot z^{-1}_{\mathsf{cnt}[w]}$ are random elements from G[3]. Thus, $y_{\mathsf{cnt}[w]}$ also leaks nothing.

4. *Together,* $((\Sigma.\mathsf{utk}, y_{\mathsf{cnt}[w]}), \mathsf{xtag})$ *at most leak* $\emptyset$. $\Sigma.\mathsf{utk}$ is generated from an independent protocol from $(y_{\mathsf{cnt}[w]}, \mathsf{xtag})$. Thus, $\Sigma.\mathsf{utk}$ together with $(y_{\mathsf{cnt}[w]}, \mathsf{xtag})$ leaks nothing. Additionally, $y_{\mathsf{cnt}[w]}$ and xtag are two random elements of the group G. So there is no leakage altogether.

This concludes that the leakage of a single update operation is $\emptyset$.

Update-Update Leakage.

1. $\Sigma.\mathsf{utk}$ *of many updates leaks* $\emptyset$. The leakage from an update of SKSSE is $\emptyset$ even when the server sees many updates. This ensures that $\Sigma.\mathsf{utk}$ of our scheme for two or more updates is also $\emptyset$.

[3] A formal argument provided in Game G_5 of Theorem 5.4.1

2. $y_{\mathsf{cnt}[w]}$ *at most leaks* $\emptyset$. For every update operation, the input $w \| \mathsf{cnt}[w]$ to $F_p^{(Z)}$ is always unique. The pseudo-randomness of F_p ensures that $y_{\mathsf{cnt}[w]}$ in several updates are indistinguishable from a random element of G.

3. xtag *at most leaks* $\emptyset$. From the above discussion, we see that for any two update operations, both x_w and xid cannot be the same. Moreover, the server never gets to see x_w or xid in clear in any operation. When the server sees many update operations, it might happen that two updates use the same x_w (when a keyword is the same) or the same xid (when id and op are the same). However, both x_w and xid would never be the same for two update operations. Hence, eddh hardness of G (Definition 2.2.3) ensures that no efficient adversary can distinguish the xtag which is $g^{x_w \cdot \mathsf{xid}}$ from a random element of G even when the server sees many xtags either with same x_w or with same xid[4].

4. *Together,* $((\Sigma.\mathsf{utk}, y_{\mathsf{cnt}[w]}), \mathsf{xtag})$ *at most leak* $\emptyset$. From the games G_1, G_5 and G_6 in the Proof of Theorem 5.4.1, we know that each of these values can be simulated independently of each other. Since the adversary cannot distinguish between any of these games (except with negligible probability), the server does not get any additional information from this tuple, other than what it does from each of its elements individually.

Search-Update Leakage. Consider the following sequence of two queries.

1. The current update query q_β at time instance β is, $q_\beta = (\beta, \mathsf{op}_\beta, (\mathsf{id}_\beta, w_\beta))$, that is input to $\Pi.\mathsf{Update}_\mathsf{C}$ and outputs $((\Sigma.\mathsf{utk}_\beta, y_\beta), \mathsf{xtag}_\beta)$, where $\mathsf{op}_\beta \in \{\mathsf{add}, \mathsf{del}\}$.
2. For $\alpha < \beta$, a previous search query $q_\alpha = (\alpha, \mathsf{srch}_\alpha, \omega_\alpha)$, where $\omega_\alpha = w_1 \wedge w_2 \wedge \cdots \wedge w_n$, that is input to $\Pi.\mathsf{Search}_\mathsf{C}$ and outputs $(\Sigma.\mathsf{stk}_\alpha, \Gamma.\mathsf{xtk}_\alpha)$.

We argue that the leakage due to the outputs of q_α and q_β in each case is no more when combined. Both $\Sigma.\mathsf{utk}_\beta$ and $\Sigma.\mathsf{stk}_\alpha$ are outputs to the server for the underlying single keyword search protocol Σ. Hence the leakage due to the combination of these

[4]The argument is similar to the argument in Game G_6 of Theorem 5.4.1.

two parameters is no more than has been accounted for in $\mathcal{L}_{\Sigma,\mathsf{Update}}$. Now, as we only invoke Σ with the add operation, there is no leakage.

In the proof of Theorem 5.4.1, we have argued in Game G_5 that the values $y_j \leftarrow F_p{}^{(I)}(\mathsf{k_I}, \mathsf{id}\|\mathsf{op}) \cdot (F_p{}^{(Z)}(\mathsf{k_Z}, w\|j))^{-1}$, are essentially random elements of G to the adversary. Due to our setting, the tuple $(\mathsf{op}, w, \mathsf{id})$ received in q_β has never appeared in any previous update. This ensures that y_β received in the current update operation when combined with xtk_α received during the search operations at time instance α, does not generate an $\mathsf{xtag} = \mathsf{xtk}_\alpha[i,j]^{y_\beta}$ that is present in XDB. Similarly, xtag_β received in q_β is also not present in XDB. Thus, xtag_β and y_β received in the current update combined with any previous search query q_α do not leak any more than the leakage after the current update. All of the above together show that the update leakage for our scheme is $\mathcal{L}_{\Pi,\mathsf{Update}} = \emptyset$.

Search Leakage

Single Search Leakage. For a conjunctive search query $\omega = w_1 \wedge w_2 \wedge \cdots \wedge w_n$, w_1 is the *s-term* and the search token is $(\Pi.\mathsf{stk}, \Pi.\mathsf{xtk})$.

1. From $\Pi.\mathsf{stk} = \Sigma.\mathsf{stk}$, the server will know $\mathsf{sp}(w_1)$, $\mathsf{TimeDB}(w_1)$ and $\mathsf{Updates}(w_1)$, as these are leakages of Σ (see Definition 2.5.6). Note that $\mathsf{DelHist}(w_1)$ shall not be leaked as we call $\Sigma.\mathsf{Update}_{\mathcal{C}}$ with $\mathsf{op} = \mathsf{add}$. The actual update operation is appended to the id. So even if the id in two updates are the same, $(\mathsf{id}\|\mathsf{op})$ is always different for the same keyword. Thus, if a keyword is added and deleted from the same identifier, the id in both cases would be different, and the server cannot link the add and delete operations.

2. Next, the server computes

$$\mathsf{xtk}[j,i]^{y_j} = g^{F_p{}^{(Z)}(\mathsf{k_Z}, w_1\|j)\cdot x_{w_i}\cdot\left(\mathsf{xid}\cdot F_p{}^{(Z)}(\mathsf{k_Z}, w_1\|j)^{-1}\right)},$$

   for all $j \in [\mathsf{cnt}[w_1]]$ and $i \in [2,n]$, using $\Pi.\mathsf{xtk}$ output by $\Gamma.\mathsf{Search}$, where $x_{w_i} = F_p{}^{(X)}(\mathsf{k_X}, w_i)$. The server then checks if the generated $\mathsf{xtag}_{i,j} = \mathsf{xtk}[j,i]^{y_j}$ is in XDB or not.

(a) If $\mathsf{xtag}_{i,j} \in \mathsf{XDB}$, the server knows that the $\mathsf{xid} = F_p(\mathsf{k_X}, \mathsf{id}\|\mathsf{op})$ that was added for the *s-term* must have also been added for the *x-term*. Thus, the server comes to know those timestamps when the pair $(\mathsf{op}, \mathsf{id})$ in the *s-term* matches other keywords in the conjunction.

(b) Otherwise, $\mathsf{xtag}_{i,j} \notin \mathsf{XDB}$. Our non-modifiability setting ensures that this event will not occur in future updates. So the $\mathsf{xtag}_{i,j}$ computed as $\mathsf{xtk}[j,i]^{y_j}$ which is not in XDB, does not convey any information other than the result of the conjunction to the server.

Thus, the whole leakage is subsumed by the leakage through $\mathsf{Updates}(\omega)$ and $\mathsf{TimeDB}(\omega)$.

Here we note that $\mathsf{DelHist}(\omega)$ is never leaked as we only add an (id, w) pair using Σ, even in case of deletion for the protocol Π. So Σ only works with $\mathsf{op} = \mathsf{add}$, and the actual operation remains hidden from Σ and hence from the server.

We note that $\mathsf{TimeDB}(\omega)$ is a huge overestimation of the actual leakage of Π. The actual leakage of $\mathsf{TimeDB}(\omega)$ in our scheme is the following.

$$\overline{\mathsf{TimeDB}}(\omega) = \mathsf{TimeDB}(w_1) \cup \{(u, \mathsf{id}) : (i \geq 2) \wedge ((\mathsf{id}, *) \in \mathsf{TimeDB}(w_1)) \wedge ((u, \mathsf{add}, (\mathsf{id}, w_i)) \in \mathsf{QS})\}.$$

In other words, the timestamps of those identifiers are leaked for *x-term*s which are only present in the *s-term*.

Search-Search Leakage. For $\alpha < \beta$, consider two search queries $q_\alpha = (\alpha, \mathsf{srch}_\alpha, \omega_\alpha)$ and $q_\beta = (\beta, \mathsf{srch}_\beta, \omega_\beta)$. Any search token $\Pi.\mathsf{xtk}$ contains $\mathsf{xtk}[j,i] = g^{F_p{}^{(Z)}(\mathsf{k_Z}, w_1\|j) \cdot F_p{}^{(X)}(\mathsf{k_X}, w_i)}$ for $j \in [\mathsf{cnt}[w_1]]$ and $i \in [2,n]$. Now say, ω_α and ω_β have the same *s-term*s. So the xtks are the same whenever w_1 and w_i are repeated in the two searches. Hence, the server learns if a pair (w_1, w_i) has appeared in multiple conjunctive search queries through the equality of xtks. This is true for any $X = w_{i_1} \wedge w_{i_2} \wedge \cdots \wedge w_{i_\ell}$, where $w_{i_j} \in \widetilde{\mathcal{L}}_\omega$ for $j \in [\ell]$. That is, for the least frequent keyword w_1 and an arbitrary conjunction

of set of keywords in $\widetilde{\mathcal{L}}(\omega)$, the server comes to know the joint-search pattern

$$\mathsf{sp}(w_1, X) = \{u | (u, \mathsf{srch}, \omega) \in \mathsf{QS}, \omega = w_1 \wedge \cdots \wedge X \wedge \cdots\}.$$

Hence, for a conjunctive query $\omega = w_1 \wedge \cdots \wedge w_n$, the total search pattern leakage of our scheme is

$$\mathsf{sp}(\omega) = \mathsf{sp}(w_1) \cup \left(\bigcup_{i=2}^{n} \mathsf{sp}(w_1, w_i) \right).$$

Update-Search Leakage. Consider a search query $q_\beta = (\beta, \mathsf{srch}_\beta, \omega_\beta)$. For $\alpha < \beta$, consider a previous update query $q_\alpha = (\alpha, \mathsf{op}_\alpha, (\mathsf{id}_\alpha, w_\alpha))$, where $\mathsf{op}_\alpha \in \{\mathsf{add}, \mathsf{del}\}$ and $\omega_\beta = w_1 \wedge w_2 \wedge \cdots \wedge w_n$. If $w_\alpha \notin \mathcal{L}(\omega_\beta)$, that is, if a keyword that was updated previously is not present in the current search query, then such update and search queries are unrelated and hence there is no additional leakage. If $w_\alpha \in \mathcal{L}(\omega_\beta)$, the leakage is precisely that captured by the single search leakage described above, as an update operation does not leak anything.

All of the above together show that the search leakage for our scheme is subsumed by the following leakage we have assumed for the proof of Theorem 5.4.1.

$$\mathcal{L}_{\Pi,\mathsf{Search}} = \left(\mathsf{sp}(\omega), \overline{\mathsf{TimeDB}}(\omega), \mathsf{Updates}(\omega)\right).$$

Leakage Profile of Π. We finally summarise the overall leakage of our CDSSE scheme. By combining the leakage of Σ and Γ, we indeed get the leakage of Π that is subsumed by the following leakage.

$$\mathcal{L}_{\Pi,\mathsf{Setup}}(\mathsf{DB}) = \emptyset, \quad \mathcal{L}_{\Pi,\mathsf{Update}}(\mathsf{op}, \mathsf{id}, w) = \emptyset,$$
$$\mathcal{L}_{\Pi,\mathsf{Search}}(\omega) = \left(\mathsf{sp}(\omega), \overline{\mathsf{TimeDB}}(\omega), \mathsf{Updates}(\omega)\right).$$

Remark 1. *As mentioned in [95], for a file injection attack [97] to work properly, the adversary must know the results of all the sub-queries of the form $w_i \wedge w_j$ for all w_i, w_j in the conjunction and $1 \leq i < j \leq n$, that is the KPRP leakage. Although our scheme does not prevent such leakage, the precise leakage of our scheme is $w_1 \wedge w_i$*

for all w_i in the conjunction. Moreover, as shown by [80], random shuffling of xtk*s reduces the success probability of the file injection attack like [97] to almost 5% even when the amount of injected files is 60% of the entire database.*

Remark 2. *It is important to note that our* xtag *is deterministic. Consider an update operation* $(\mathsf{op}, (\mathsf{id}, w_1))$ *while* $(\mathsf{op}, (\mathsf{id}, w_i))$ *has not occurred yet. The* xtag *that would be generated for the search* $\omega = w_1 \wedge \cdots \wedge w_i \wedge \cdots w_n$ *would not be present in the* XDB*. Now, if at a later stage, the update* $(\mathsf{op}, (\mathsf{id}, w_i))$ *indeed happens, then the* xtag *for that update would be exactly the same* xtag *that server has computed at the time of the search operation* $\omega = w_1 \wedge \cdots \wedge w_i \wedge \cdots w_n$*. Thus, the server would be able to link the update operation* $(\mathsf{op}, (\mathsf{id}, w_i))$ *with the search operation* $\omega = w_1 \wedge \cdots \wedge w_i \wedge \cdots w_n$*, breaking the forward privacy of* Π*. The above event resembles that at the time of adding the document identifier* id*,* w_1 *was present in that document and at a later stage* w_i *was added to the document. However, if an update* $(\mathsf{op}, (\mathsf{id}, w_1))$ *has occurred and,* $(\mathsf{op}, (\mathsf{id}, w_i))$ *has not, in that case, our setting of non-modifiable documents ensures that this operation would never happen in any future update, preventing such kind of forward privacy breach. The above is the reason why* Π *is not secure in the general case of SSE. This was missed by [80].*

Remark 3. *From the concrete leakage analysis it is quite evident that, even if our construction can be instantiated with a* BPIP *secure SKSSE, the resulting CDSSE is not* BPIP *secure. The reason is that even if we store the update for every keyword with a* BPIP *secure scheme, the* xtag*s are stored in a set membership query data structure. Now, at the time of a search operation, when an* xtag *generated for any keyword in the conjunction is found in the set, it reveals the time when the element was added to the set, degrading the whole security to* BPUP*. However, if our scheme is instantiated using a* WBP *secure single keyword construction, the security is enhanced to* BPUP*. The reason is, in the update phase we always call our single keyword SSE with* $\mathsf{op} = \mathsf{add}$*, and the actual operation is embedded with the* id *(as we replace* id *with* $\mathsf{id}\|\mathsf{op}$*). So the server is unaware of the actual operation as the server never sees the original identifiers. Thus the server cannot link the add operation to the delete*

operation as it cannot distinguish between these two operations. Achieving BPIP secure CDSSE is still an open problem.

5.6 Implementation and Experimental Results

We implemented three instantiations of our generic construction with three different DSSEs [87, 49, 38]. Our code is available at [77]. We conducted experiments on the Enron Email database[45] to evaluate the performances of these three implementations. All three schemes performed very well. Our implementation proved to be very practical and scalable.

Implementation Details. We have instantiated our generic scheme with three different SKSSE schemes – one of which is only forward private [87], and two are state-of-the-art forward and backward private schemes [49] and [38]. The schemes were implemented in C++. The code is available at [77]. The basic codes for these experiments were written by Theo Henault.

All the experiments were performed on a desktop computer with Intel Core i7-6700 CPU, 3.40 GHz with 8 Cores and 16GB memory. The database was stored using RocksDB [44], and all cryptographic primitives were implemented using Crypto++[5]. We report our results in a similar fashion as reported in [29, 80, 100]. We have used the Enron Email Dataset[45] for our experiments. In [87, 38], when two consecutive queries have the same keyword, the server stores the previous search result to optimise the search. However, for a fair comparison of the time taken by the three schemes, we have not implemented this optimisation – all times reported are for fresh search of the *s-term*s. We do not report our search results with different probabilities of search operation. However, the modification suggested in [87, 38] can also be applied for single keyword searches, which will improve the search performance of the corresponding implementations.

[5]Available at: https://www.cryptopp.com/

EDB Creation and Update Operation. The EDB was created using the Enron Email Dataset. We wrote a C++ program to extract IDs and keywords from the dataset. We got $517,401$ IDs and $72,252,961$ ID-keyword pairs. Table 5.2 shows the sizes of the encrypted databases and clients' state sizes for different schemes. We also report the total time and the time per pair, required to create the EDB. The EDB creation essentially calls the update process of our protocol (without the network delay), so the average update time per pair was calculated while generating the EDB and is reported in Table 5.2. The update time for an id-keyword pair is almost constant in all three instantiations. We have performed an independent experiment by inserting 10^4 pairs and taking the average time. The experiment also matches the result reported in Table 5.2.

Schemes	Time (Hr.)	Time/Pair (ms)	Server Storage (GB)	Client Storage (MB)
FAST-OXT	12.6	0.63	15.5	20
Mitra-OXT	10.4	0.52	9.1	2
Π_{BP}-OXT	11.1	0.55	13.3	3.1

Table 5.2: Storage cost and time required for each update.

Search Operation. To evaluate the performance of search operations, we follow the same setting as in [80], and [100]. We report the time for search queries that are conjunctions of two keywords ($w_1 \wedge w_2$). We have conducted two types of search operations,

1. In the first experiment, we keep the size of the number of updates performed on the second keyword to a constant $|\mathsf{Updates}(w_2)| = 10^5$ and vary $|\mathsf{Updates}(w_1)|$ from 100 to 10^5 (Figure 5-4). The exact values are in Table 5.3
2. In the second case, we keep the size of the number of updates performed on the first keyword to a constant, that is, $|\mathsf{Updates}(w_1)| = 10$ and vary $|\mathsf{Updates}(w_2)|$ from 10 to 10^5 (Figure 5-5).

In both cases, we report the search time. The time required to search two keywords both having order of 10^5 updates takes ~ 80 sec for FAST-OXT, ~ 60 sec for Mitra-OXT and ~ 52 sec for Π_{BP}-OXT (Figure 5-4). Our results concur with the results

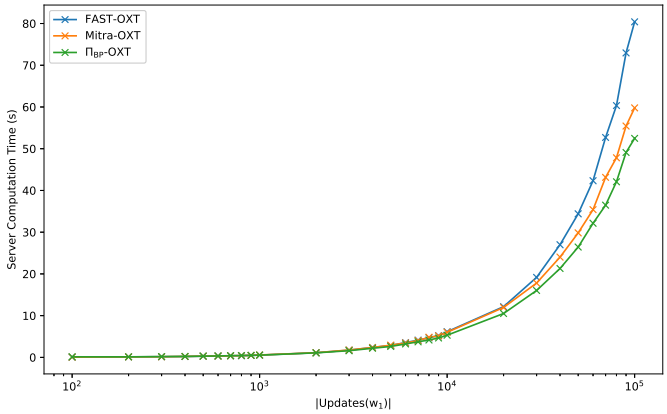


Figure 5-4: Search time comparison between the three instantiations of our generic CDSSE for fixed $|\text{Updates}(w_2)| = 10^5$ with varying $|\text{Updates}(w_1)|$ from 10^2 to 10^5.

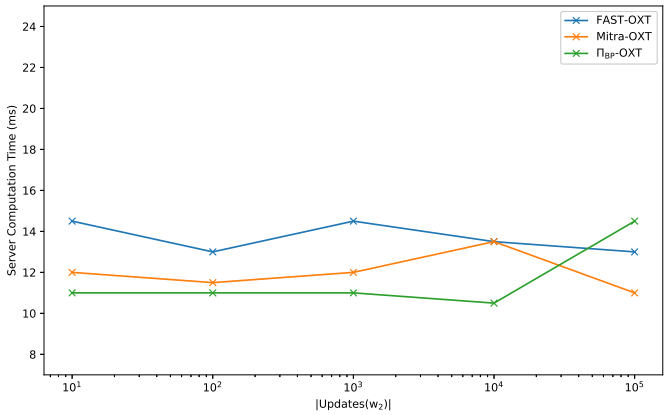


Figure 5-5: Search time comparison between the three instantiations of our generic CDSSE for fixed $|\text{Updates}(w_1)| = 10$ with varying $|\text{Updates}(w_2)|$ from 10 to 10^5.

from [38], as Mitra-OXT outperforms Π_{BP}-OXT when the search output size is small and Π_{BP}-OXT outperforms Mitra-OXT for large search output sizes. The search time is high for FAST-OXT as decryption is done at the server.

We did not perform searches with more than 2 keywords in conjunction as the search for elements on XDB can be *parallelised* and hence will essentially take similar time as a conjunctive query with 2 keywords.

The work of [100] uses an SSE to store the entries of *s-term* and *x-term* as well. The implementation of [100] is not available. However, as reported in their paper, they did their experiment on a Ubuntu 20.04.3 LTS workstation with Intel @Xeon(R) W-2123 CPU 3.60GHz with 8 cores and 32GB RAM. Their setup uses more process-

ing power with a double-sized RAM. As reported in their paper, their protocol has a higher running time when the number of updates for both w_1 and w_2 is high. This is understandably so, as for each *s-term* and *x-term* they have to access an SSE. However, their running time overshoots our scheme by a huge margin when the number of updates for the *s-term* is small but the number of updates for the *x-term* is high. This is because we only fetch very few *x-term*s (precisely the number of times the *s-term* has been updated) whereas they have to fetch all the *x-term*s related to the keyword. Similarly, as reported in [95], they only test their scheme for a database with 23,643 documents, 60,879 keywords, and 8,373,977 keyword-document pairs, which is almost a magnitude smaller than our test case. Their server machines were running Ubuntu 18.04 LTS and had 16 cores (Intel Core i9-9900 CPU 3.10 GHz), 31GB RAM, and 483 GB SSD disk space. However, their running time is also similar to our results when the matching search output size is small and their search time grows rapidly when the matching search output size increases, making their scheme much less suitable for large databases compared to ours.

$\lvert\mathsf{Updates}(w_1)\rvert$	Mitra-OXT (s)	FAST-OXT (s)	Π_{BP}-OXT (s)
10^2	0.06	0.06	0.13
10^3	0.57	0.54	0.53
10^4	6.01	6.15	5.31
10^5	59.79	80.41	52.49

Table 5.3: Search time comparison for fixed $\lvert\mathsf{Updates}(w_2)\rvert = 10^5$ with varying $\lvert\mathsf{Updates}(w_1)\rvert$ from 10^2 to 10^5

5.7 Final Remarks

We proposed a generic dynamic searchable symmetric encryption scheme that allows conjunctive queries while being both forward and backward private. Our scheme is specifically crafted to protect *non-modifiable* documents that, once uploaded, their keywords remain unaltered. This is a common characteristic in a variety of applications, including biometric databases, digital archives, legal documents, medical records, etc. Our construction allows the keyword set associated with a document

to be modified with some extra overhead. With efficiency closest to OXT (among known dynamic schemes) and by ensuring both forward and backward privacy, our proposed construction offers a viable solution for preserving the privacy of dynamic datasets while facilitating conjunctive queries.

The OXT framework has been extended [29] to support general boolean queries and to the multi-client setting [57] in a modular black-box fashion. Both these extensions facilitate a lot of practical use cases. Our construction uses OXT in a black-box fashion as well, where the functionality of OXT has not been altered. Hence, the extension of our generic scheme to support general boolean queries and multi-clients seems quite plausible for future work.

6

Updatable Message Authentication Codes

Our next goal is to construct a fault-tolerant, verifiable DSSE. The main tool that we use for our construction is a message authentication code with some added functionality which we call an updatable message authentication code (UdMAC). In this Chapter, we systematically develop UdMACs, and in the next chapter, we use them to construct a fault-tolerant verifiable DSSE.

Message authentication codes (MACs) are algorithms that have been traditionally used to provide integrity of a message. A MAC scheme is a pair of algorithms (*MAC generation algorithm, Verification algorithm*). On a given message and a key *MAC generation algorithm* outputs a *tag*. The sender sends this *tag* along with the message. Given a message, the key and a *tag*, the *Verification algorithm* outputs either 0 or 1, where 1 indicates that the message is authentic and 0 indicates the message is not authentic. There are several paradigms for constructing MACs. For our purpose, we need to extend the basic functionality of a MAC. We consider a scenario where a client outsources its message M to be stored in a server. To ensure the integrity of the message, the client computes a MAC t for the message and transmits M along with t to be stored in the server. Over time, if the client needs to update M, then (s)he has to download the message M update it, re-compute the MAC and then send it back to the server. If the message is large and frequent updates are required, this naive solution may be inefficient. We propose a new type of MAC, which we call *updatable MAC* (UdMAC), which has the property that for updates, the client does not need to download the message and recompute the MAC. Suppose the message

M is to be updated by m and we denote the updated message by $M\Delta m$; the client should be able to compute the update MAC t_u for $M\Delta m$ using only m and t. Thus, the client only sends the update m along with t_u to the server, and the server stores the updated message $M\Delta m$ and the updated tag t_u.

This specialized functionality provided by UdMACs is not available in traditional MACs. We define the syntax and security of UdMACs, and provide two constructions, namely, ConCatU and XoRU. The two constructions work for two different types of updates, concatenation and xor difference, respectively. We analyze in detail both the constructions and prove their security.

There is a long history of cryptographic schemes with updating capabilities starting from the work reported in [12, 13], where the concepts of incremental collision resistant hash functions and signatures were introduced. These ideas were further refined and formalized in [10]. Another line of work focuses on key updatable encryption. A key updatable encryption scheme is a symmetric encryption scheme that allows the key holder to update keys and to compute an update token, which can be given to a party storing ciphertexts, and can be used to update existing ciphertexts to ones under the new key [19, 18, 46]. Studies on key updatable signatures and message authentication codes have been reported in [39]. In [39] the key updatable MACs are called updatable MACs (UMAC). We do not consider key updates, but updates in the message. We also call our MACs updatable MACs but use a different acronym, UdMAC. Our work aligns more with the scenarios considered in [12, 13, 10] but our formulations and constructions are different.

6.1 Message Authentication Codes

A message authentication code (MAC) is a map $F : \mathcal{K} \times \mathcal{M} \to \{0,1\}^\tau$, where $\mathcal{K}$ is the key space and $\mathcal{M}$ the message space. We often write $F_K(\cdot)$ to denote $F(K, \cdot)$. The output of a MAC is called the tag, and τ is called the tag length. The security of a MAC F is defined using an interaction of F with an adversary $\mathcal{A}$. It is assumed that $\mathcal{A}$ has an oracle access to $F_K(\cdot)$, where $K \xleftarrow{\$} \mathcal{K}$. For a query $x \in \mathcal{M}$ of $\mathcal{A}$ the oracle

responds by sending $y = F_K(x)$. Let, $\mathcal{A}$ queries $x_1, x_2, \ldots, x_q$ and gets $y_1, y_2, \ldots, y_q$ as responses from the oracle. These queries are performed adaptively. Finally, $\mathcal{A}$ outputs a pair (x^*, y^*), where $x^* \notin \{x_1, x_2, \ldots, x_q\}$. This pair is called a forgery and it is said that $\mathcal{A}$ has successfully *forged* F if $F_K(x^*) = y^*$. The auth-advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}_F^{\mathsf{auth}}(\mathcal{A}) = \Pr[K \xleftarrow{\$} \mathcal{K} : \mathcal{A} \text{ forges}].$$

We say that F is (ϵ, t) secure if for every adversary $\mathcal{A}$, which runs for time at most t, $\mathsf{Adv}_F^{\mathsf{auth}}(\mathcal{A}) \leq \epsilon$. It is well known [20] that for any arbitrary adversary $\mathcal{A}$ for the MAC F there exists a PRF adversary $\mathcal{B}$ for F such that

$$\mathsf{Adv}_F^{\mathsf{auth}}(\mathcal{A}) \leq \mathsf{Adv}_F^{\mathsf{prf}}(\mathcal{B}) + \frac{1}{2^\tau}, \tag{6.1}$$

where $\mathcal{B}$ and $\mathcal{A}$ both run almost for the same time and ask almost the same number of queries.

6.2 Polynomial MACs

Let $\mathbb{F}_{2^n}$ denote the field with 2^n elements, n-bit strings can be seen as elements in $\mathbb{F}^{2^n}$. Thus additions and multiplication of n-bit strings are addition and multiplications in $\mathbb{F}_{2^n}$. For $x, y \in \{0,1\}^n$, we denote their addition and multiplication by $x \oplus y$ and xy respectively.

For $x \in \{0,1\}^*$ and $h \in \{0,1\}^n$, let $(x_1, x_2, \ldots, x_m) = \mathsf{parse}(x)$. We define the function

$$\mathsf{Poly}_h(x) = x_1 h \oplus x_2 h^2 \oplus \cdots \oplus \mathsf{Pad}_n(x_m) h^m$$

where the additions and multiplications are in $\mathbb{F}_{2^n}$ and for $y \in \{0,1\}^*$, $|y| \leq n$, $\mathsf{Pad}_n(y) = y \| 0^{n-|y|}$.

A related function is

$$\mathsf{PPoly}_h(x) = \mathsf{Poly}_h(x) \oplus |x| h^{m+1}.$$

The following result is well known:

Proposition 6.2.1. *Let $x_1, x_2 \in \{0,1\}^*$, $x_1 \neq x_2$ and let $\ell_1 = \lceil |x_1|/n \rceil$ and $\ell_2 = \lceil |x_2|/n \rceil$ then for any $c \in \{0,1\}^n$*

$$\Pr[h \xleftarrow{\$} \{0,1\}^n : \mathsf{PPoly}_h(x_1) \oplus \mathsf{PPoly}_h(x_2) = c] \leq \frac{\max\{\ell_1, \ell_2\} + 1}{2^n}.$$

The nonce based Wegman Carter MAC [26, 92] is a widely used MAC algorithm. A popular variant of this MAC takes in a message $x \in \{0,1\}^*$, two keys $K, h \in \{0,1\}^n$ and a nonce $N \in \{0,1\}^n$ and produces an n-bit tag as

$$\mathsf{WC}_{K,h}(x; N) = \mathsf{PPoly}_h(x) \oplus E_K(N), \tag{6.2}$$

where $E_K()$ is a pseudo-random function family.

The Theorem below regarding the security of Wegman Carter MAC is well known (for example see [20]).

Theorem 6.2.1. *Let Wegman Carter MAC be as defined in Equation (6.2) and $\mathcal{A}$ be a nonce respecting[1] forgery adversary who asks at most q queries. Among the q queries, let the message queried of the largest length be of length ℓ. Then there exists a PRF adversary $\mathcal{B}$ for $E_K()$, such that*

$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{WC}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{prf}}_{E}(\mathcal{B}) + \frac{\ell + 1}{2^n}.$$

We will need the following result regarding the function $\mathsf{Poly}_h()$.

Proposition 6.2.2. *Let $x \in \{0,1\}^{n\ell_1}$ and $y \in \{0,1\}^{n\ell_2}$, and let $(x_1, x_2, \ldots, x_{\ell_1}) = \mathsf{parse}_n(x)$ and $(y_1, y_2, \ldots, y_{\ell_2}) = \mathsf{parse}_n(y)$ and $x_{\ell_1} \neq 0^n$ and $y_{\ell_2} \neq 0^n$ and $x \neq y$. Then, for any $c \in \{0,1\}^n$,*

$$\Pr[h \xleftarrow{\$} \{0,1\}^n : \mathsf{Poly}_h(x) \oplus \mathsf{Poly}_h(y) = c] \leq \frac{\max\{\ell_1, \ell_2\}}{2^n}.$$

[1] A nonce respecting adversary is an adversary who never repeats a nonce in its queries.

Proof. Without loss of generality, let $\ell_1 \geq \ell_2$. Thus, we have

$$\begin{aligned} \mathsf{Poly}_h(x) &= x_1 h \oplus x_2 h^2 \oplus \cdots \oplus x_{\ell_2} h^{\ell_2} \oplus \cdots \oplus x_{\ell_1} h^{\ell_1} \\ \mathsf{Poly}_h(y) &= y_1 h \oplus y_2 h^2 \oplus \cdots \oplus y_{\ell_2} h^{\ell_2}. \end{aligned}$$

If $\ell_1 = \ell_2$ we have

$$\mathsf{Poly}_h(x) \oplus \mathsf{Poly}_h(y) = (x_1 \oplus y_1)h \oplus (x_2 \oplus y_2)h^2 \oplus \cdots \oplus (x_{\ell_1} \oplus y_{\ell_1})h^{\ell_1},$$

which is a non zero polynomial of degree at most ℓ_1 as $x \neq y$. If $\ell_1 > \ell_2$ we have

$$\mathsf{Poly}_h(x) \oplus \mathsf{Poly}_h(y) = (x_1 \oplus y_1)h \oplus (x_2 \oplus y_2)h^2 \oplus \cdots \oplus (x_{\ell_2} \oplus y_{\ell_2})h^{\ell_2} \oplus \cdots \oplus x_{\ell_1} h^{\ell_1},$$

which is a non zero polynomial of degree at most ℓ_1 as $x_{\ell_1} \neq 0^n$.

Thus, in both cases we have $\mathsf{Poly}_h(x) \oplus \mathsf{Poly}_h(y) \oplus c = 0$ have at most ℓ_1 roots in $\mathbb{F}_{2^n} = \{0,1\}^n$.

Similar argument holds if $\ell_2 > \ell_1$ and in that case we have $\mathsf{Poly}_h(x) \oplus \mathsf{Poly}_h(y) \oplus c = 0$ have at most ℓ_2 roots in $\mathbb{F}_{2^n} = \{0,1\}^n$. Thus we have

$$\Pr[h \xleftarrow{\$} \{0,1\}^n : \mathsf{Poly}_h(x) \oplus \mathsf{Poly}_h(y) = c] \leq \frac{\max\{\ell_1, \ell_2\}}{2^n}.$$

□

6.3 Updatable MACs

An updatable MAC works on four associated sets $\mathcal{K}$, $\mathcal{M}$, Σ and $\mathcal{T}$ called the key space, message space, state space and tag space, respectively. We will consider $\mathcal{M} \subset \{0,1\}^*$ and an associative binary operation $\Delta : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, which we will further call as an update function. An updatable MAC Ψ is a triple of algorithms $\Psi = (\mathsf{KeyGen}, \mathsf{MacUpdate}, \mathsf{MacVerify})$ described as follows:

$\mathsf{KeyGen}(1^\lambda)$: takes as input the security parameter λ outputs a key K, selected

uniformly at random from the key space $\mathcal{K}$.

$\mathsf{MacUpdate}(K, m, \sigma)$: takes as input $K \in \mathcal{K}$, $m \in \mathcal{M}$ and $\sigma \in \Sigma \cup \{\perp\}$ and outputs a $t \in \mathcal{T}$ and an updated state $\widetilde{\sigma}$.

$\mathsf{MacVerify}(K, m, \sigma, t)$: takes in the key K, a $m \in \mathcal{M}$, $\sigma \in \Sigma$ and $t \in \mathcal{T}$ and outputs a bit $b \in \{0, 1\}$.

Correctness: Let $m_1, m_2, \ldots, m_s \in \mathcal{M}$ be a sequence of messages and $\Psi = (\mathsf{KeyGen}, \mathsf{MacUpdate}, \mathsf{MacVerify})$. For an updatable MAC Ψ, we define a procedure $\Psi.\mathsf{SeqUpdt}$ in Figure 6-1 which takes in a key $K \in \mathcal{K}$ generated by $\Psi.\mathsf{KeyGen}(1^\lambda)$, a sequence of messages $(m_1, m_2, \ldots, m_s)$ and outputs $(\sigma, t) \in \Sigma \times \mathcal{T}$. We say that Ψ is correct, if for any message sequence $(m_1, \ldots, m_s)$ and any K generated by $\Psi.\mathsf{KeyGen}$,

$$\Psi.\mathsf{MacVerify}(K, M, \Psi.\mathsf{SeqUpdt}(K, (m_1, m_2, \ldots, m_s))) = 1,$$

where $M = m_1 \ \Delta \ m_2 \ \Delta \ \cdots \ \Delta \ m_s$.

$\Psi.\mathsf{SeqUpdt}(K, (m_1, m_2, \ldots, m_s))$
01. $(t_1, \sigma_1) \leftarrow \Psi.\mathsf{MacUpdate}(K, m_1, \perp)$;
02. **for** $i = 2$ to s,
03. $\quad (\sigma_i, t_i) \leftarrow \Psi.\mathsf{MacUpdate}(K, m_i, \sigma_{i-1})$;
04. **end for**
05. **return** (σ_s, t_s);

Figure 6-1: The procedure $\mathsf{SeqUpdt}$ used to define correctness of an updatable MAC scheme Ψ.

The definition of correctness specifies the usefulness of updatable MAC. Consider the $\mathsf{MacUpdate}$ procedure is applied on the messages $m_1, m_2, \ldots, m_s$, one after the other in the same sequence as in the procedure $\mathsf{SeqUpdt}$ of Figure 6-1. Thus, we start with the message m_1 and an empty state and in the subsequent messages the input to $\mathsf{MacUpdate}$ is the current message and the state obtained for the previous

message. The final state-tag pair (σ_s, t_s) obtained should be a valid MAC for the message $M = m_1 \,\Delta\, m_2 \,\Delta\, \cdots \,\Delta\, m_s$.

6.3.1 Security of Updatable MACs

For defining security of an updatable MAC $\Psi = (\mathsf{KeyGen}, \mathsf{MacUpdate}, \mathsf{MacVerify})$, we give $\mathcal{A}$ oracle access of $\Psi.\mathsf{MacUpdate}(K, \cdot, \cdot)$, instantiated with a key K generated by $\Psi.\mathsf{KeyGen}(1^\lambda)$. $\mathcal{A}$ can query its oracle with $(m, \sigma) \in \mathcal{M} \times \Sigma$ and get as response $(\widetilde{\sigma}, t) \in \Sigma \times \mathcal{T}$. Let $\mathbb{Q} = \{(m_1, \sigma_1), (m_2, \sigma_2), \ldots (m_q, \sigma_q)\}$ be the set of queries made by $\mathcal{A}$ and for the i^{th} query (m_i, σ_i), let the response of the oracle be $(\widetilde{\sigma}_i, t_i)$. We denote the set of messages queried by $\mathcal{A}$ by $\mathbb{M} = \{m : (m, \sigma) \in \mathbb{Q}\}$ and we call the set

$$\mathbb{P}_{\mathcal{A}} = \{(\sigma_i, m_i, \widetilde{\sigma}_i, t_i) : 1 \leq i \leq q\}$$

as the query profile of the adversary $\mathcal{A}$. Finally, after $\mathcal{A}$ stops querying, it outputs a forgery (M^*, σ^*, t^*).

We call a forgery (M^*, σ^*, t^*) of an adversary $\mathcal{A}$ with query profile $\mathbb{P}_{\mathcal{A}}$ as *"invalid"* if there exists a sequence of query-responses $(\sigma^{(1)}, m^{(1)}, \widetilde{\sigma}^{(1)}, t^{(1)}), (\sigma^{(2)}, m^{(2)}, \widetilde{\sigma}^{(2)}, t^{(2)}), \ldots,$ $(\sigma^{(\ell)}, m^{(\ell)}, \widetilde{\sigma}^{(\ell)}, t^{(\ell)})$, such that the following are true simultaneously

1. $\{(\sigma^{(1)}, m^{(1)}, \widetilde{\sigma}^{(1)}, t^{(1)}), (\sigma^{(2)}, m^{(2)}, \widetilde{\sigma}^{(2)}, t^{(2)}), \ldots, (\sigma^{(\ell)}, m^{(\ell)}, \widetilde{\sigma}^{(\ell)}, t^{(\ell)})\} \subseteq \mathbb{P}_{\mathcal{A}}$.
2. $m^{(1)} \,\Delta\, m^{(2)} \,\Delta\, \cdots \,\Delta\, m^{(\ell)} = M^*$
3. $\sigma^{(1)} = \perp$, $\widetilde{\sigma}^{(\ell)} = \sigma^*$ and $\sigma^{(i)} = \widetilde{\sigma}^{(i-1)}$ for $1 \leq i \leq \ell$.

A forgery (M^*, σ^*, t^*) is valid if it is not invalid. An adversary $\mathcal{A}$ is successful if its forgery (M^*, σ^*, t^*) is valid and $\Psi.\mathsf{MacVerify}(M^*, \sigma^*, t^*) = 1$. Let $\mathsf{Succ}(\mathcal{A})$ be the event that $\mathcal{A}$ is successful. We define the forging advantage of an adversary $\mathcal{A}$ for a Δ-updatable MAC as

$$\mathsf{Adv}_F^{\mathsf{Uauth}}(\mathcal{A}) = \Pr[\mathsf{Succ}(\mathcal{A})]. \tag{6.3}$$

The probability is taken over the selection of the key K with which the oracle is instantiated, the (possible) randomness in Ψ and the (possible) randomness in $\mathcal{A}$.

6.3.2 Some New Notations

In the subsequent sections we describe two constructions MACs which support two different update functions. The first one is called ConCatU which converts any given PRF $f : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ to an $\|$-updatable MAC, i.e., where the update function is concatenation. The second construction converts a Wegman Carter MAC [26] to an $\oplus$-updatable MAC, i.e., a UdMAC where the update function is xor differences.

We introduce a bit of new notation for convenience. Let $(\mathbb{N})^*$ denote the set of all lists whose elements are natural numbers including the empty list. Thus a finite list $L \in (\mathbb{N})^*$ can be seen as an array, and its i^{th} element will be denoted by $L[i]$, and its length, i.e., the number of elements present in it, will be denoted by $||L||$. An empty list is denoted by $\langle\rangle$. For an $a \in \mathbb{N}$, $\mathsf{append}(L, a)$ appends a at the end of L. By $\mathsf{sum}(L)$ we will mean the sum of all the elements in L. Let, $L \in (\mathbb{N})^*$ be a finite length list of length $||L|| = \ell$ where $L = \langle x_1, x_2, \ldots, x_\ell \rangle$ and let $M \in \{0,1\}^*$ be a string of length $\mathsf{sum}(L)$. Then, $\mathsf{Parse}(M, L)$ returns $(m_1, m_2, \ldots, m_l)$ such that $|m_i| = L[i]$ and $m_1 \| m_2 \| \ldots \| m_l = M$. For a list $L = \langle x_1, x_2, \ldots, x_\ell \rangle$, by $\mathsf{str}(L)$ we mean the $n\ell$ bit string $\mathsf{bin}_n(x_1) \| \mathsf{bin}_n(x_2) \| \cdots \| \mathsf{bin}_n(x_\ell)$. If $L = \langle\rangle$, $||L|| = 0$ and $\mathsf{str}(L)$ is the empty string.

6.4 ConCatU: An Updatable MAC for Concatenation

Let $M \subset \{0,1\}^*$, $\mathcal{K} = \{0,1\}^k \times \{0,1\}^n$, $\Sigma \subset (\mathbb{N})^* \times \{0,1\}^n \cup \{\perp\}$, $\mathcal{T} = \{0,1\}^n$ be the message space, key space, state space and tag space of ConCatU respectively. Let $f : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ be a PRF family. We use f to construct a Δ-updatable MAC ConCatU, where Δ is the concatenation function $\|$.

A state σ of ConCatU is either $\perp$ or it contains two distinct fields denoted as $\sigma = (L_\sigma, \mathsf{pTag}_\sigma)$, where L_σ is a finite list of positive integers, and pTag_σ is a n-bit binary string.

$\mathsf{ConCatU.KeyGen}()$ outputs a pair (K, h), where $K \xleftarrow{\$} \{0,1\}^k$ and $h \xleftarrow{\$} \{0,1\}^n$.

The procedures ConCatU.MacUpdate and ConCatU.MacVerify are shown in Figure 6-2.

To understand the workings of the procedures ConCatU.MacUpdate and ConCatU.MacVerify it is important see the role of the state σ.

Consider an initial message m_1, and let subsequent updates on it be $m_2, \ldots, m_k$. As the update function is a concatenation function, hence the message after the i^{th} update, $1 \leq i \leq k$, will be $M_i = m_1 \| m_2 \| \cdots \| m_i$. We see the initial message also as an update on an empty message. The intended usage of ConCatU.MacUpdate is to obtain a correct authentication tag of the updated message after each update without knowing the whole message. To accomplish this, each update m that is given as an input to the ConCatU.MacUpdate procedure is associated with a state σ. The state σ carries some information regarding the message on which the current update m is to be applied. In particular, σ keeps a record of the history of updates on the message. The procedure produces an authentication tag for the updated message and also updates the state with the information of the current update.

For an input (m, σ), if $\sigma = \perp$, then it signifies m is the initial message, i.e., an update m on an empty message is being sought. The ConCatU.MacUpdate procedure on such an input should produce a MAC tag for the message m, which in this case is the updated message. Further, the procedure should populate the initially empty list L with the length of m and set the current state to (L, tag) and output it along with the tag.

When an input (m, σ) is received with $\sigma \neq \perp$, then a MAC is being sought for the message $M \| m$, where M is some message which is not being explicitly provided to the algorithm. But, the information regarding M is encoded in the state $\sigma = (L, \mathsf{pTag})$, which carries the following information:

1. The message M, over which the update is being sought was obtained after $k = ||L||$ updates.

2. The initial update on the empty message, i.e., the initial message was of length $L[1]$ bits.

3. For $2 \leq i \leq k$, the i^{th} update increased the length of the message by $L[i]$ bits.

Thus, for a correct usage, $|M| = \mathsf{sum}(L)$.

4. pTag was the tag computed by $\mathsf{ConCatU.MacUpdate}$ for the message M, i.e., the last call to $\mathsf{ConCatU.MacUpdate}$ for this specific message returned pTag.

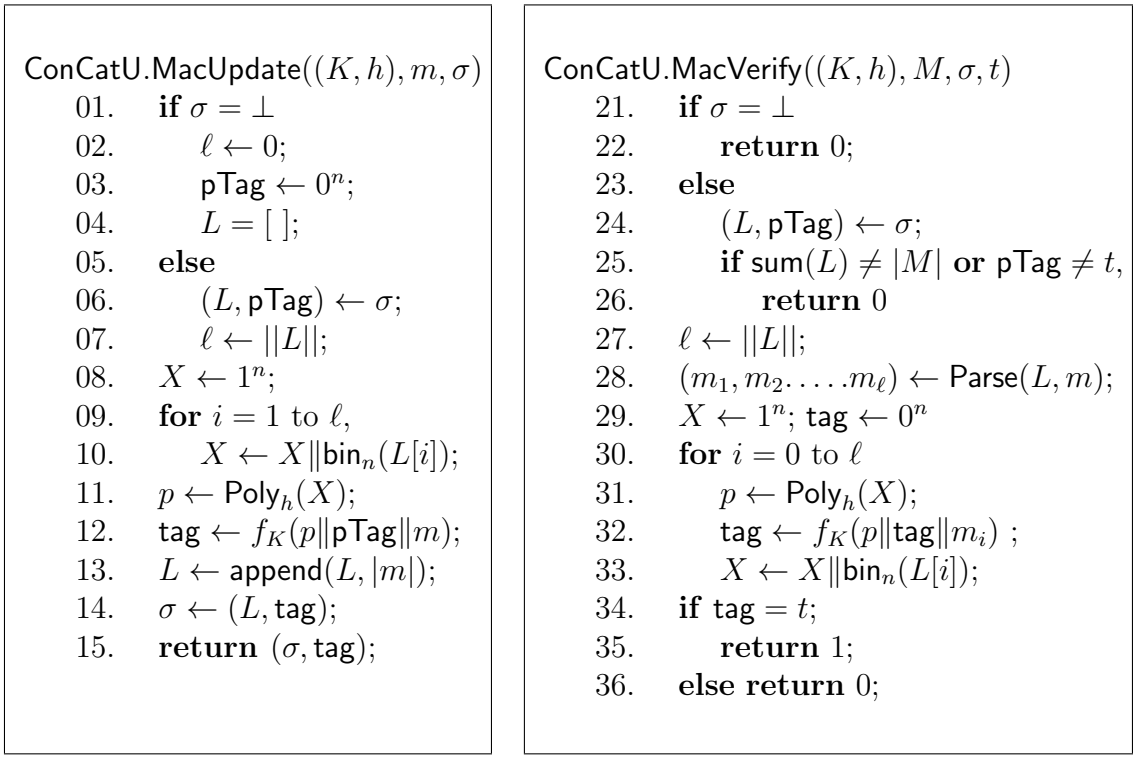


Figure 6-2: Specification of $\mathsf{ConCatU}$ using a prf $f_K : \{0,1\}^* \to \{0,1\}^n$.

As specified in Figure 6-2, the update procedure $\mathsf{ConCatU.MacUpdate}$ on an input (m, σ), parses $\sigma = (L, \mathsf{pTag})$ if $\sigma \neq \perp$ and if $\sigma = \perp$ it sets L to an empty list and pTag to 0^n. Further, it computes a string X as the concatenation of the n-bit binary encodings of the entries in L. In particular, if $\sigma = \perp$, then X is set to 1^n and otherwise it is set as $X = 1^n \| \mathsf{bin}_n(L[1]) \| \cdots \| \mathsf{bin}_n(L[\ell]) = 1^n \| \mathsf{str}(L)$, where $\ell = ||L||$. Then, p is computed as $\mathsf{Poly}_h(X) \oplus |X| h^{\ell+2}$ and finally the tag is produced as $f_K(p \| \mathsf{pTag} \| m)$. The new state σ is constructed as $(\mathsf{append}(L, |m|), \mathsf{tag})$.

The verification process takes a message M, a state σ and a tag t. First, it does some checks to verify if the state σ and the message M are compatible. In particular, if the following are true, then the verification procedure rejects by returning 0.

1. $\sigma = \perp$. This is an incompatible state for any message for verification, as $\sigma = \perp$ signifies that the message has never gone through the ConCatU.MacUpdate procedure.

2. $\sigma \neq \perp$ and $\sigma = (L, \mathsf{pTag})$ but $\mathsf{sum}(L) \neq |M|$. This also represents an incompatible state for the message M. As the list L keeps track of the updates performed on a message. The size of the list $||L||$ represents the number of updates that have taken place and $L[i]$ denotes the number of bits concatenated to the message in the i-th update. Hence, if $\mathsf{sum}(L) \neq |M|$, then σ represents an incompatible state for M.

3. $\sigma \neq \perp$ and $\sigma = (L, \mathsf{pTag})$ but $\mathsf{pTag} \neq t$. This is incompatible, as the ConCatU.MacUpdate always writes the current computed tag in the pTag field of σ.

After these checks of compatibility, it sets $\ell = ||L||$, and parses the message M as $(m_1, m_2, \ldots, m_\ell)$ such that $M = m_1 \| m_2 \| \cdots \| m_\ell$ and $|m_i| = L[i]$. Then, in lines 29 to 33, it basically runs the procedure $\mathsf{ConCatU.SeqUpdt}((K, h), (m_1, m_2, \ldots, m_\ell))$ as described in Figure 6-1, i.e., it sequentially computes updates of $m_1, m_2, \ldots, m_\ell$ starting from an empty message. Finally, if the computed tag matches the input tag t, then it accepts by returning 1, else it rejects by returning 0. Based on the verification procedure described in Figure 6-2, it is easy to verify that ConCatU is a correct UdMAC.

6.4.1 Instantiations and Efficiency

We have motivated UdMACs to be used in client-server scenarios, where a client stores a message M along with a tag t in an un-trusted server and it requires updating the message. In the context of ConCatU the allowed update is concatenation. The client wants to compute the authentication tag for the updated message without downloading the previous message from the server. We will view the procedures of ConCatU in this context as this would help us to evaluate the efficiency of the procedure.

A motivating use case may be the following. Suppose the client records real time video footage through a surveillance camera and every hour uploads the raw video data to a server. Thus, every hour, the message in the server is updated by concatenation and by using $\mathsf{ConCatU}$, one can compute the tag using just the data of one hour without needing any access to the whole data.

In such a client server scenario. If a single message is being updated multiple times, it is best for the client to store the state of the message with itself. In the case of $\mathsf{ConCatU}$, the state contains the tag and the list of updates. The size of the state for a message which has undergone k updates would be $n(k+1)$ bits, n-bits for the tag and nk bits for the list L. For our example, the size of the state grows just n-bits per hour, whereas the size of the message grows by about a GB per hour (this would depend on the resolution of the video etc). Note, n being the tag length, an acceptable value for it would be just 128.

Storing the state, along with some extra information with the client would have efficiency implications while computing the tag. As per the procedure depicted in Figure 6-2, an update with a message m and state (L, pTag) requires computing $p = \mathsf{Poly}_h(X)$, and one computation of $f_k(p\|\mathsf{pTag}\|m)$. The other computations required are not significant. Note, $X = 1^n\|\mathsf{str}_n(L)$ and thus $|X| = (||L|| + 1)n$, hence $\mathsf{Poly}_h(X)$ is of degree at most $(||L|| + 1)$, and computing $\mathsf{Poly}_h(X)$ will require $(||L|| + 1)$ additions and multiplications in the field $\mathbb{F}_{2^n}$. But, the client may keep a bit more information to speed up this computation. Along with the state, it also stores the current value of p (see line 11 of Figure 6-2) for each update. When a message m_1 is first uploaded then p is computed as $(1^n)h$ and L is initialized as a list containing a single element, where $L[1] = |m_1|$. The client stores both p and L. When m_1 is updated with m_2, then instead of recomputing p the client just updates it as $p = (p + |m_2|)h$, and uses this value in 12. This update incurs just one multiplication and one addition. This can be continued, and each update would require just one multiplication and one addition for computing p irrespective of the current state of the message.

In addition to p, the client needs to compute $f_k(p\|\mathsf{pTag}\|m)$. The only require-

ment for $f_k()$ is that it has to be a variable input length PRF. Thus, $f_k()$ can be instantiated with any block-cipher based secure deterministic message authentication code like PMAC [16, 81], PAuth [32], secure variants of CBC-MAC [15, 56, 74, 62] etc. The parallelizable MACs like PMAC and PAuth are particularly suited as they are efficient across several platforms and require just $\lceil |M|/n \rceil$ many block-cipher calls to authenticate a message M. Thus, a single update call for a message m will require $\lceil (|m|+2)/n \rceil \approx \lceil |m|/n \rceil + 2$ block-cipher calls and one multiplication in $\mathbb{F}_{2^n}$.

6.4.2 Security of ConCatU

First, let us consider a small modification in the procedure ConCatU.MacUpdate where the procedure only returns the tag, i.e., it does not output the updated state, though it computes it. We call this modified construction as ConCatU.MU1. Our first observation is that if f is a PRF, then ConCatU.MU1 is also a PRF. Before we formalize this fixing a notation would be useful.

Definition 6.4.1. *A (q, r, ℓ)-adversary is an adversary who asks q queries in the query phase where the queries are (σ_i, m_i), $1 \le i \le q$, each $\sigma_i = (L_i, \mathsf{pTag}_i) \in (\mathbb{N})^* \times \{0,1\}^n$ and $r = \max_i\{||L_i||\}$ and $\ell = \max_i\{|m_i|\}$. Note if $\sigma = \perp$ we consider $\sigma = (\langle\rangle, 0^n)$.*

Proposition 6.4.1. *Let $\mathcal{A}$ be an arbitrary (q, r, ℓ)-PRF adversary for* ConCatU.MU1. *Then there exists a* PRF *adversary $\mathcal{B}$ for f_K such that*

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{ConCatU.MU1}}(\mathcal{A}) \le \mathsf{Adv}^{\mathsf{prf}}_{f}(\mathcal{B}) + \binom{q}{2}\frac{r}{2^n}.$$

Proof. We assume that $\mathcal{A}$ does not repeat queries. As it is a (q, r, ℓ)-PRF adversary, it asks q distinct queries.

Consider the i-th query (σ_i, m_i) of $\mathcal{A}$. If $\sigma_i = \perp$ we set $L_i = \langle\rangle$, $X_i = 1^n$ and $\mathsf{pTag}_i = 0^n$. If $\sigma_i \neq \perp$ then $\sigma_i = (L_i, \mathsf{pTag}_i)$, and we set $X_i = 1^n \| \mathsf{str}_n(L_i)$. For the i-th query of $\mathcal{A}$ we define the string $Z_i = p_i \| \mathsf{pTag}_i \| m_i$, where

$$p_i = \mathsf{Poly}_h(X_i).$$

Note that on the i-th query, the function $f_K()$ is called with input Z_i. Let the set of all inputs to $f_K()$ during the queries of the adversary be

$$\mathbb{S} = \{Z_i : 1 \leq i \leq q\}.$$

Let us define the following events:

G0: $\mathcal{A}$ interacting with ConCatU.MU1 instantiated with $K \xleftarrow{\$} \{0,1\}^k$ and $h \xleftarrow{\$} \{0,1\}^n$ and outputting a 1.

G1: $\mathcal{A}$ interacting with ConCatU.MU1, where $f_K()$ is replaced by a uniform random function ρ drawn from the set of all functions mapping $\{0,1\}^*$ to $\{0,1\}^n$, and outputting 1.

G2: $\mathcal{A}$ interacting with ConCatU.MU1, where $f_K()$ is replaced by \$(), which on any input returns a uniform random n bit string, and outputting a 1. Note that ConCatU.MU1 where $f_K()$ is replaced by \$() essentially outputs a n-bit random string irrespective of its input.

COLL: While interacting with ConCatU.MU1, where $f_K()$ is replaced by a uniform random function ρ, $\mathcal{A}$ asks two distinct queries $i, j \in [q]$, $i \neq j$ such that $Z_i = Z_j$.

By definition of PRF advantage, we have

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{ConCatU.MU1}}(\mathcal{A}) = |\Pr[\mathsf{G0}] - \Pr[\mathsf{G2}]|\,. \tag{6.4}$$

By a standard reduction, we can construct an adversary $\mathcal{B}$ such that

$$|\Pr[\mathsf{G0}] - \Pr[\mathsf{G1}]| \leq \mathsf{Adv}^{\mathsf{prf}}_{f}(\mathcal{B}), \tag{6.5}$$

Also, the events G1 and G2 are same unless the event COLL occurs, hence by the difference lemma,we have

$$\Pr[\mathsf{G1}] - \Pr[\mathsf{G2}]| \leq \Pr[\mathsf{COLL}]. \tag{6.6}$$

Using the above equations, we have the following sequence of inequalities.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{ConCatU.MU1}}(\mathcal{A}) &= |\Pr[\mathsf{G0}] - \Pr[\mathsf{G2}]| \\
&= |(\Pr[\mathsf{G0}] - \Pr[\mathsf{G1}]) + (\Pr[\mathsf{G1}] - \Pr[\mathsf{G2}])| \\
&\leq |(\Pr[\mathsf{G0}] - \Pr[\mathsf{G1}])| + |(\Pr[\mathsf{G1}] - \Pr[\mathsf{G2}]| \\
&\leq \mathsf{Adv}^{\mathsf{prf}}_{f}(\mathcal{B}) + \Pr[\mathsf{COLL}].
\end{aligned}$$

We are only left to bound $\Pr[\mathsf{COLL}]$. For which we consider two distinct queries (σ_i, m_i), (σ_j, m_j) for which $\rho()$ gets called with Z_i and Z_j respectively. We consider the following cases:

Case 1. $m_i \neq m_j$. In this case $\Pr[Z_i = Z_j] = 0$.

Case 2. $m_i = m_j$. As $(\sigma_i, m_i) \neq (\sigma_j, m_j)$, thus $\sigma_i \neq \sigma_j$. We have to consider two sub cases here. If $\mathsf{pTag}_i \neq \mathsf{pTag}_j$, then $\Pr[Z_i = Z_j] = 0$. If $\mathsf{pTag}_i = \mathsf{pTag}_j$, then $L_i \neq L_j$. Which implies $X_i \neq X_j$ and by Proposition 6.2.2 we have $\Pr[Z_i = Z_j] = \Pr[p_i = p_j] \leq (\max(||L_i||, ||L_j||))/2^n$.

Thus, by the above two cases, the fact that $\mathbb{S}$ contains q elements, and the union bound, we have

$$\Pr[\mathsf{COLL}] \leq \binom{q}{2}\frac{r}{2^n}. \tag{6.7}$$

This completes the proof. □

The following Theorem claims security of $\mathsf{ConCatU}$ against a forgery adversary.

Theorem 6.4.1. *Let $\mathcal{A}$ be an arbitrary (q, r, ℓ)-forgery adversary attacking $\mathsf{ConCatU}$, also let the forgery produced by $\mathcal{A}$ be (σ, m, t), where $\sigma = (L, \mathsf{ptag})$, where $||L|| = r^*$. Then there exists a PRF adversary $\mathcal{B}$ for f such that*

$$\mathsf{Adv}^{\mathsf{Uauth}}_{\mathsf{ConCatU}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{prf}}_{f}(\mathcal{B}) + \frac{1 + (q+1)r^*}{2^n} + \frac{q\max\{r, r^*\}}{2^n} + \binom{q}{2}\frac{r}{2^n}.$$

Proof. We denote the real scheme instantiated with $(K, h) \xleftarrow{\$} \{0,1\}^k \times \{0,1\}^n$ by $\mathsf{ConCatU}(f_K, h)$. By $\mathsf{ConCatU}(\rho, h)$ we denote the update and verification algorithms

of $\mathsf{ConCatU}$, where the function $f_K()$ is replaced by a uniform random function ρ drawn from the set of all functions mapping $\{0,1\}^*$ to $\{0,1\}^n$.

We call the event of $\mathcal{A}$ attacking $\mathsf{ConCatU}(f_K, h)$ and producing a valid forgery which passes verification by $\mathsf{Succ}^{(f_K,h)}(\mathcal{A})$. Similarly, we call the event of $\mathcal{A}$ attacking $\mathsf{ConCatU}(\rho, h)$ and producing a valid forgery which passes verification by $\mathsf{Succ}^{(\rho,h)}(\mathcal{A})$.

By an easy reduction, we can construct a PRF adversary $\mathcal{B}$ such that

$$\mathsf{Adv}^{\mathsf{Uauth}}_{\mathsf{ConCatU}}(\mathcal{A}) = \Pr[\mathsf{Succ}^{(f_K,h)}(\mathcal{A})] \leq \mathsf{Adv}^{\mathsf{prf}}_{f}(\mathcal{B}) + \Pr[\mathsf{Succ}^{(\rho,h)}(\mathcal{A})]. \tag{6.8}$$

Now we try to bound $\Pr[\mathsf{Succ}^{(\rho,h)}_{\mathcal{A}}]$.

We assume that $\mathcal{A}$ does not repeat any query and makes q distinct queries. Let its i^{th} query be (σ_i, m_i), and for such a query it gets a response $(\widetilde{\sigma}_i, t_i)$. Thus, the query profile of $\mathcal{A}$ is

$$\mathbb{P}_{\mathcal{A}} = \{(\sigma_i, m_i, \widetilde{\sigma}_i, t_i) : 1 \leq i \leq q\}.$$

Also, we denote

$$\mathbb{Q}_{\mathcal{A}} = \{(\sigma_i, m_i) : 1 \leq i \leq q\},$$

the set of distinct queries made by $\mathcal{A}$.

As in the proof of Proposition 6.4.1, consider the i-th query (σ_i, m_i). If $\sigma_i = \perp$ we set $L_i = \langle\rangle$, $X_i = 1^n$ and $\mathsf{pTag}_i = 0^n$. If $\sigma_i \neq \perp$ then $\sigma_i = (L_i, \mathsf{pTag}_i)$, and we set $X_i = 1^n \| \mathsf{str}_n(L_i)$. For the i-th update query of $\mathcal{A}$ we define the string $Z_i = p_i \| \mathsf{pTag}_i \| m_i$, where

$$p_i = \mathsf{Poly}_h(X_i).$$

Note that on the i-th query, the random function ρ is called on the string Z_i. Let the set of all inputs to ρ during the queries of the adversary be

$$\mathbb{S} = \{Z_i : 1 \leq i \leq q\}.$$

Claim 6.4.1. *Let* $\mathsf{E0}$*, be the event that there is a collision in the set* $\mathbb{S}$*. Then,*

$$\Pr[\mathsf{E0}] \leq \binom{q}{2}\frac{r}{2^n}. \tag{6.9}$$

As the event E_0 is exactly the event COLL defined in the proof of Proposition 6.4.1, hence Equation (6.7) proves the above claim.

Let the valid forgery produced by $\mathcal{A}$ be (σ^*, m^*, t^*). It is required that $\sigma^* \neq \perp$, as otherwise the success probability of $\mathcal{A}$ will be zero. Let $\sigma^* = (L^*, \mathsf{pTag}^*)$, where $L^* = \langle \ell_1^*, \ell_2^*, \ldots, \ell_{r^*}^* \rangle$, and

$$\ell_1^* + \ell_2^* + \cdots + \ell_{r^*}^* = \ell^* = |m^*|.$$

Also, let $(m_1^*, m_2^*, \ldots, m_{r^*}^*) = \mathsf{Parse}(m^*, L^*)$. Also define $L_0^* = \langle\rangle$ and for $1 \leq i \leq r^*$, $L_i^* = \mathsf{append}(L_{i-1}^*, \ell_i^*)$.

For ease of exposition, we define some variables. For $0 \leq i \leq r^*$ we define

$$X^{(i)} = \begin{cases} 1^n & \text{if } i = 1 \\ X^{(i-1)} \| \mathsf{bin}_n(\ell_{i-1}) & \text{otherwise,} \end{cases} \tag{6.10}$$

and for $1 \leq j \leq r^*$

$$p^{(j)} = \mathsf{Poly}_h(X^{(j-1)}).$$

For $0 \leq i \leq r^*$

$$\mathsf{tag}^{(i)} = \begin{cases} 0^n & \text{if } i = 0 \\ f_K(p^{(i)} \| \mathsf{tag}^{(i-1)} \| m_i^*) & \text{otherwise,} \end{cases} \tag{6.11}$$

and for $1 \leq i \leq r^*$,

$$Y^{(i)} = p^{(i)} \| \mathsf{tag}^{(i-1)} \| m_i^*.$$

and let $\mathbb{Y} = \{Y^{(i)} : 1 \leq i \leq r^*\}$. Note that while verification of the forgery produced by $\mathcal{A}$, ρ gets evaluated on the strings in $\mathbb{Y}$.

Now, we define a set of queries $\mathbb{Q} = \{Q_i : 1 \leq i \leq r^*\}$, where

$$Q_i = \begin{cases} (\perp, m_i^*) & \text{if } i = 1 \\ ((L_{i-1}^*, \mathsf{tag}^{(i-1)}), m_i^*) & \text{if } i > 1. \end{cases}$$

Note that, if Q_i is given as an update query, then the function ρ gets evaluated on the string $Y^{(i)}$. Also this query set $\mathbb{Q}$ is defined by the forgery attempt of $\mathcal{A}$.

As the forgery produced by $\mathcal{A}$ is a valid forgery, hence according to the definition of a valid forgery, $\mathcal{A}$ has not asked the queries $Q_1, Q_2, \ldots, Q_{r^*}$ in the query phase in the same order.

Definition 6.4.2. *Given the forgery of $\mathcal{A}$ the query set $\mathbb{Q}_{\mathcal{A}}$ is called out of order, if there exists a $i \in [r^* - 1]$, such that query $\mathbb{Q}_{i+1}$ was asked before query $\mathbb{Q}_i$.*

Definition 6.4.3. *Given the forgery of $\mathcal{A}$ the query set $\mathbb{Q}_{\mathcal{A}}$ is called incomplete, if there exists a $i \in [r^*]$, such that query $\mathbb{Q}_i \notin \mathbb{Q}_{\mathcal{A}}$.*

Thus for the forgery produced by $\mathcal{A}$ to be valid its query set $\mathbb{Q}_{\mathcal{A}}$ has to be either out of order or incomplete.

Claim 6.4.2. *Let* $\mathsf{E1}$ *be the event that the query set $\mathbb{Q}_{\mathcal{A}}$ is out of order. Then,*

$$\Pr[\mathsf{E1}] \leq \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n}. \tag{6.12}$$

Proof of Claim: Let for $i \in [q-1]$ the query Q_{i+1} was asked before Q_i. For these queries, the random function ρ was evaluated on Z_{i+1} and Z_i respectively in the same order. Assuming that there was no collision on $\mathbb{S}$, we have $Z_i \neq Z_{i+1}$. Also,

$$\begin{aligned} Z_{i+1} &= p_i + 1 \| \mathsf{pTag}_{i+1} \| m_{i+1}, \\ Z_i &= p_i \| \mathsf{pTag}_i \| m_i, \end{aligned}$$

and $\mathsf{pTag}_{i+1} = \rho(Z_i)$. As all Z_i's are distinct thus $\mathcal{A}$ has not seen the output of $\rho(Z_i)$ before it asked the query Q_i, as ρ is a random function. Thus the probability that $\mathcal{A}$ could have asked the query $Q_{i+1} = ((L_i, t), m_i)$ where $t = \rho(Z_i)$ is at most $1/2^n$.

Hence using the union bound over all possible $i \in [r^*]$ which are out of order, we have $\Pr[\mathsf{E1}|\mathsf{E0}] \leq r^*/2^n$. Hence,

$$\begin{aligned} \Pr[\mathsf{E1}] &\leq \Pr[\mathsf{E1}|\mathsf{E0}] + \Pr[\mathsf{E0}] \\ &\leq \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n}. \end{aligned} \tag{6.13}$$

This completes the proof of the claim.

Now, we assume that the query set $\mathbb{Q}_{\mathcal{A}}$ is not out of order, then if the forgery of $\mathcal{A}$ is valid, then $\mathbb{Q}_{\mathcal{A}}$ must be incomplete, i.e., there exists a $j^* \in [r^*]$ such that Q_{j^*} is not in $\mathbb{Q}_{\mathcal{A}}$. Thus we have the following claim.

Claim 6.4.3. *For $j^* \leq k \leq r^*$, let $\mathsf{E2}^{(k)}$ be the event that $Y^{(k)} \in \mathbb{S}$. Then*

$$\Pr[\mathsf{E2}^{(j^*)}] \leq \frac{q \max\{r, j^*\}}{2^n} + \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n}. \tag{6.14}$$

and for $j^* + 1 \leq k \leq r^*$,

$$\Pr[\mathsf{E2}^{(k)}|\neg\mathsf{E2}^{(k-1)}] \leq \frac{q}{2^n}. \tag{6.15}$$

For a proof of Equation (6.14), notice that, if $Q_{j^*} = ((L_j^*, \mathsf{pTag}_{j^*}), m_{j^*})$ was never asked in the query phase then $\Pr[Y^{(j^*)} = Z_i]$ for some $i \in [q]$ is at most $\max\{r, j^*\}/2^n$. This follows from Proposition 6.2.2. Thus as there are q many Z_i's in $\mathbb{S}$, by the union bound we have, $\Pr[\mathsf{E2}^{(j^*)}|\neg\mathsf{E1}]$ is at most $q \max\{r, j^*\}/2^n$. Hence,

$$\begin{aligned} \Pr[\mathsf{E2}^{(j^*)}] &\leq \Pr[\mathsf{E2}^{(j^*)}|\neg\mathsf{E1}] + \Pr[\mathsf{E1}] \\ &\leq \frac{q \max\{r, j^*\}}{2^n} + \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n}. \end{aligned}$$

For Equation (6.15), we observe that if $Y^{(k-1)}$, not in $\mathbb{S}$, then $\mathsf{tag}^{(k-1)}$ is a n-bit uniform random string, and thus $Y^{(k)} = p^{(k)}\|\mathsf{tag}^{(k-1)}\|m_k^*$ is equal to any $Z_i \in \mathbb{S}$ with probability at most $1/2^n$, and hence

$$\Pr[\mathsf{E2}^{(k)}|\neg\mathsf{E2}^{(k-1)}] \leq \frac{q}{2^n}.$$

as claimed.

Finally, we would like to bound the probability of success of $\mathcal{A}$, i.e., we need to bound $\Pr[\mathsf{Succ}_{\mathcal{A}}^{(\rho,h)}]$.

$$\begin{aligned}\Pr[\mathsf{Succ}_{\mathcal{A}}^{(\rho,h)}] &= \Pr[t^* = \mathsf{tag}^{(r^*)}] \\ &\leq \Pr[t^* = \mathsf{tag}^{(r^*)} | \neg \mathsf{E2}^{(r^*)}] + \Pr[\mathsf{E2}^{(r^*)}] \\ &\leq \frac{1}{2^n} + \Pr[\mathsf{E2}^{(r^*)}]. \end{aligned} \tag{6.16}$$

The last inequality is due to the fact that if $Y^{(r^*)} \notin \mathbb{S}$, then $\mathsf{tag}^{(r^*)}$ is a uniform random n-bit string and t^* is a fixed string fixed by $\mathcal{A}$ before $\rho()$ was evaluated on $Y^{(r^*)}$.

Now, by using repeated conditioning, we have

$$\begin{aligned}\Pr[\mathsf{E2}^{(r^*)}] &= \sum_{k=0}^{r^*-j^*} \Pr[\mathsf{E2}^{(r^*-k)} | \neg \mathsf{E2}^{(r^*-(k-1))}] + \Pr[\mathsf{E2}^{(j^*)}] \\ &\leq \frac{(r^*-j^*)q}{2^n} + \frac{q \max\{r, j^*\}}{2^n} + \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n} \\ &< \frac{r^*q}{2^n} + \frac{q \max\{r, r^*\}}{2^n} + \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n}. \end{aligned} \tag{6.17}$$

Finally using Equations (6.8),(6.16) and (6.17), we get

$$\begin{aligned}\Pr[\mathsf{Succ}_{\mathcal{A}}^{(f_K,h)}] &\leq \mathsf{Adv}_f^{\mathsf{prf}}(\mathcal{B}) + \frac{1}{2^n} + \frac{r^*q}{2^n} + \frac{q \max\{r, r^*\}}{2^n} + \frac{r^*}{2^n} + \binom{q}{2}\frac{r}{2^n} \\ &= \mathsf{Adv}_f^{\mathsf{prf}}(\mathcal{B}) + \frac{1 + (q+1)r^*}{2^n} + \frac{q \max\{r, r^*\}}{2^n} + \binom{q}{2}\frac{r}{2^n}. \end{aligned}$$

□

6.5 XoRU: An Updatable MAC for $\oplus$ Difference

Here, we discuss a scheme for UdMAC where the update function is $\oplus$, i.e., on a message M, if an update m is applied then it results in the message $M \oplus m$. We always assume that the length of the updated message is at least the size of the

original message. In particular, it is important to note the following points:

1. If the current message is M and the intended updated message is M', where $|M| = |M'|$, then the update request should be with the string $m = M \oplus M'$, and here $|m| = |M|$.

2. If the current message is M and the intended updated message is M' where $|M| < |M'|$, then an update request should be issued with $m = M \| 0^{|M'|-|M|}$. In this case $|m| = |M'|$.

3. If the current message is M and the intended updated message is M' where $|M| > |M'|$, such an update is not allowed. It is possible to modify the construction of XoRU that we present next to allow such kinds of updates, but it will require some extra information to be stored in the state. We do not further explore this here, as such updates will not be required in the context of the verifiable SSE that we design in the next Chapter.

Let $x \in \{0,1\}^*$, and $x_1, x_2, \ldots, x_m = \mathsf{parse}_n(x)$. For $h \in \{0,1\}^n$, define

$$\mathsf{Poly}_h(x) = x_1 h \oplus x_2 h^2 \oplus \cdots \oplus \mathsf{Pad}_n(x_m) h^m$$

where the multiplications are in $\mathbb{F}_{2^n}$.

For XoRU we consider the key space to be $\mathcal{K} = \{0,1\}^k \times \{0,1\}^n$, the state space to be $\Sigma \cup \{\perp\}$, where σ is either $\perp$ or can be parsed as $\sigma = (\mathsf{pNonce}, \mathsf{pLen}, \mathsf{pTag})$, where $\mathsf{pNonce}, \mathsf{pLen}, \mathsf{pTag} \in \{0,1\}^n$, and $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ is a fixed input length PRF.

The MacUpdate and Verify procedures for XoRU are described in Figure 6-3. Next we note down some elementary observations regarding the construction.

Proposition 6.5.1. *Let* $m_1, m_2, \ldots, m_s \in \mathcal{M}$ *and* $\mathsf{XoRU.SeqUpdt}((K,h), m_1, m_2, \ldots, m_s)) = (\sigma_s, t_s)$. *Let* $\sigma_s = (\mathsf{N}, \mathsf{L}, \mathsf{T})$. *Then*

1. $\mathsf{L} = \max(|m_1|, |m_2|, \ldots, |m_s|)$

2. $t_s = \mathsf{T} = \mathsf{Poly}_h(m_1 \oplus m_2 \oplus \cdots \oplus m_s) \oplus \mathsf{L} h^{\lceil \mathsf{L}/n \rceil} \oplus E_K(\mathsf{N})$.

$\mathsf{XoRU.MacUpdate}(K, \sigma, N, m)$	$\mathsf{XoRU.MacVerify}(K, \sigma, m, t)$
01. **if** $\sigma = \perp$ **then**	21. **if** $\sigma = \perp$
02. $\quad \mathsf{tag} \leftarrow \mathsf{Poly}_h(m) \oplus \lvert m \rvert h^{\lceil \lvert m \rvert / n \rceil} \oplus E_K(N)$;	22. $\quad$ **return** 0;
03. **else**	23. **end if**
04. $\quad (\mathsf{pNonce}, \mathsf{pLen}, \mathsf{pTag}) \leftarrow \sigma$;	24. $(\mathsf{pNonce}, \mathsf{pLen}, \mathsf{pTag}) \leftarrow \sigma$;
05. $\quad \ell_1 = \lceil \frac{\mathsf{pLen}}{n} \rceil$;	25. **if** $\mathsf{pLen} \neq \lvert m \rvert$,
06. $\quad \ell_2 = \lceil \frac{\lvert m \rvert}{n} \rceil$;	26. **return** 0;
07. $\quad \mathsf{tag} \leftarrow \mathsf{pTag} \oplus h^{\ell_1}\mathsf{pLen} \oplus E_K(\mathsf{pNonce})$;	27. $\ell = \lceil \frac{\mathsf{pLen}}{n} \rceil$;
08. $\quad$ **if** $\mathsf{pLen} = \lvert m \rvert$,	28. $\mathsf{tag} \leftarrow \mathsf{Poly}_h(m) \oplus h^{\ell}\mathsf{pLen} \oplus E_K(\mathsf{pNonce})$;
09. $\quad\quad \mathsf{tag} \leftarrow \mathsf{tag} \oplus \mathsf{Poly}_h(m) \oplus h^{\ell_1}\mathsf{pLen} \oplus E_K(N)$;	29. **if** $\mathsf{tag} \neq t$,
10. $\quad$ **else**	30. $\quad$ **return** 0;
11. $\quad\quad \mathsf{tag} \leftarrow \mathsf{tag} \oplus \mathsf{Poly}_h(m) \oplus h^{\ell_2}\lvert m \rvert \oplus E_K(N)$;	31. **else**
12. $\quad$ **end if**	32. $\quad$ **return** 1;
13. **endif**	33. **end if**
14. $\sigma \leftarrow (N, \max\{\mathsf{pLen}, \lvert m \rvert\}, \mathsf{tag})$;	
15. **return** (σ, tag);	

Figure 6-3: Specification of XoRU.MacUpdate and XoRU.MacVerify.

Note that, point (2) in Proposition 6.5.1 essentially gives the correctness of XoRU.

6.5.1 Efficiency of XoRU

The bulk computation for update takes place in lines 07 and line 09 or line 11 of the procedure described in Figure 6-3. We first give a general estimate of the computation required.

First, we observe that if the condition in line number 08 of Figure 6-3 is satisfied then the computation of $h^{\ell_1}\mathsf{pLen}$ in both lines 07 and 10 can be avoided. As the updated tag would be computed as

$$\mathsf{tag} = \mathsf{pTag} \oplus \mathsf{Poly}_h(m) \oplus E_K(\mathsf{pNonce}) \oplus E_K(N).$$

Thus, an update satisfying the condition in Line 08 will require the computation of $\mathsf{Poly}_h(m)$, which is just an evaluation of a degree ℓ_2 polynomial with no constant term. If the Horners rule is used for evaluating the polynomial, it will take ℓ_2 multiplications and $\ell_2 - 1$ additions in $\mathbb{F}_{2^n}$. In addition to the computation of $\mathsf{Poly}_h(m)$ three more additions in $\mathbb{F}_{2^n}$ and two calls to the PRF $E_K()$ (which can be instantiated with a block-cipher like AES). would be required. If we assume the cost of one block-cipher call to be T_{AES} and the cost of one multiplication and one addition in $\mathbb{F}_{2^n}$ to be T_{add}

and T_{mult} respectively, then the total cost if $\mathsf{pLen} > |m|$ (the condition in line 08) will be

$$T_{\mathsf{add}}(\ell_2 + 2) + T_{\mathsf{mult}}\ell_2 + 2T_{\mathsf{AES}}. \tag{6.18}$$

Now, let us analyze the case where the condition in line 08 is not satisfied, i.e., we have $\mathsf{pLen} < |m|$. Note, in this case computation in line 07 will require two additions in $\mathbb{F}_{2^n}$, a computation of h^{ℓ_1} and a multiplication in $\mathbb{F}_{2^n}$. If the square and multiply algorithm for exponentiation is used then computation of h^{ℓ_1} will require at most $\log(\ell_2)$ squarings and $\log(\ell_2)$ multiplications in $\mathbb{F}_{2^n}$. Though squares in fields of characteristic 2 can be computed much more efficiently than multiplications in various platforms, for simplicity, we consider the cost of squaring to be the same as that of multiplication. With such an assumption line 07 can be computed with $2\log(\ell_1)$ multiplications. For the computation in line 11, $\mathsf{Poly}_h(m) \oplus h^{\ell_2}|m|$ can be computed with ℓ_2 additions and $\ell_2 + 1$ multiplications in $\mathbb{F}_{2^n}$. Thus summarizing, the total cost will be

$$(\ell_2 + 4)\, T_{\mathsf{add}} + (2\log(\ell_2) + \ell_2 + 1)\, T_{\mathsf{mult}} + 2T_{\mathsf{AES}}. \tag{6.19}$$

A Special Case: XoRU can be used to support the concatenation of messages by proper update requests. For example, if the current message is M and the intended message is $M\|m$, then M can be updated with an update request of $0^{|M|}\|m$. The SSE scheme which we discuss in the next chapter will require such kinds of updates and in that case both ConCatU and XoRU can be used.

Note, for each update, if $|m| = kn$ where k is a small constant, then the client can store h^{ℓ_1} where $\ell_1 = \lceil \mathsf{pLen}/n \rceil$ for each updated message, and also store $\mathsf{Poly}(M)$ for the current message M. This will require just $(2k + 2)$ multiplications for computing each updated tag and also for updating h^{ℓ_1} (as ℓ_1 increases by k in each update), and the polynomial of the current message.

6.5.2 Security of XoRU

We claim security of XoRU against a *nonce respecting* adversary, i.e., for the mac-update queries, an adversary never repeats a nonce.

For technical convenience, we consider a little change in the construction of XoRU.MacUpdate, where the procedure returns only the tag instead of (σ, tag). We call this construction as $\mathsf{XoRU.MacUpdate}^{(1)}$. This change has no effect on security as the updated state σ can be constructed by the adversary.

We define the rnd-advantage for a nonce respecting adversary $\mathcal{A}$ attacking XoRU as the following:

$$\mathsf{Adv}^{\mathsf{rnd}}_{\mathsf{XoRU}}(\mathcal{A}) = \left|\Pr[K, h \xleftarrow{\$} \{0,1\}^n : \mathcal{A}^{\mathsf{XoRU.MacUpdt}^{(1)}}(K, h, \cdot, \cdot, \cdot) = 1] - \Pr[\mathcal{A}^{\$(\cdot,\cdot,\cdot)} = 1]\right|,$$

where the $\$(\cdot,\cdot,\cdot)$ oracle on invocation with any $(\sigma, N, m) \in \Sigma \cup \{\perp\} \times \{0,1\}^n \times \mathcal{M}$ returns a uniform random n-bit string.

We claim that an efficient nonce-respecting adversary interacting with the XoRU.MacUpdate oracle instantiated with keys $(K, h) \xleftarrow{\$} \{0,1\}^n \times \{0,1\}^n$ cannot distinguish it from the oracle $\$$ if $E_K : \{0,1\}^n \leftarrow \{0,1\}^n$ is a pseudorandom function family. In other words,

Proposition 6.5.2. *For any efficient nonce respecting* rnd *adversary* $\mathcal{A}$ *attacking* XoRU*, there exists an (almost) equally efficient PRF adversary* $\mathcal{B}$ *for the function family* $E_K : \{0,1\}^n \to \{0,1\}^n$*, such that*

$$\mathsf{Adv}^{\mathsf{rnd}}_{\mathsf{XoRU.MacUpdate}}(\mathcal{A}) = \mathsf{Adv}^{\mathsf{prf}}_{E}(\mathcal{B}).$$

The proof of the above proposition is an easy reduction. The main observation that leads to the claim is that if $E_K()$ in the construction of $\mathsf{XoRU.MacUpdate}^{(1)}$ is replaced by a uniform random function then the procedure essentially outputs uniform random strings.

Now, we are in a position to state the security of XoRU.

Theorem 6.5.1. *Let* $\mathcal{A}$ *be any efficient forging adversary for* XoRU *who asks* q *many*

update queries and finally outputs a valid forgery (M^*, σ^*, t^*)*, then there exists an (almost) equally efficient PRF adversary* $\mathcal{B}$ *attacking* E_K *such that*

$$\mathsf{Adv}^{\mathsf{Uauth}}_{\mathsf{XoRU}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{prf}}(\mathcal{B}) + \left(\frac{|M^*|+1}{n}\right)\frac{1}{2^n} + \frac{1}{2^n}.$$

Proof. Let $\mathsf{XoRU}(K, h)$ be the real update and verification algorithms described in Figure 6-3 instantiated with uniform random n-bit keys K, h. Let $\mathsf{XoRU}(\rho, h)$ denote the update and verification algorithms where $E_K()$ in Figure 6-3 is replaced by a function ρ sampled uniformly at random from the set of all functions from $\{0,1\}^n$ to $\{0,1\}^n$.

We call the event of $\mathcal{A}$ attacking $\mathsf{XoRU}(K, h)$ and producing a valid forgery which passes verification by $\mathsf{Succ}^{(K,h)}_{\mathcal{A}}$. Similarly, we call the event of $\mathcal{A}$ attacking $\mathsf{XoRU}(\rho, h)$ and producing a valid forgery which passes verification by $\mathsf{Succ}^{(\rho,h)}_{\mathcal{A}}$.

By an easy reduction, we can construct a PRF adversary $\mathcal{B}$ such that

$$\mathsf{Adv}^{\mathsf{Uauth}}_{\mathsf{XoRU}}(\mathcal{A}) \;=\; \Pr[\mathsf{Succ}^{(K,h)}_{\mathcal{A}}] \leq \mathsf{Adv}^{\mathsf{prf}}(\mathcal{B}) + \Pr[\mathsf{Succ}^{(\rho,h)}_{\mathcal{A}}]. \tag{6.20}$$

Now we try to bound $\Pr[\mathsf{Succ}^{(\rho,h)}_{\mathcal{A}}]$.

First, note that according to Proposition 6.5.2, when $\mathcal{A}$ asks update queries to $\mathsf{XoRU}(\rho, h)$ (s)he gets as response random strings in place of the real tags. Thus, in the query phase, irrespective of the queries it asks, $\mathcal{A}$ gets no information regarding the real scheme.

Finally $\mathcal{A}$ outputs a valid forgery (M^*, σ^*, t^*). We consider $\sigma^* = (N^*, L^*, \mathsf{tag}^*)$. We have the following cases to consider:

Case I: There was no query with the nonce N^*: Consider the verification procedure described in Figure 6-3 where $E_K()$ is replaced by $\rho()$. Thus, in this case, in the verification procedure, $\rho(N^*)$ would be a uniform random string. Thus, the tag computed in line 27 of the procedure described in Figure 6-3 will in turn be a uniform random string. Thus, as t^* is a fixed sting, $\Pr[\mathsf{tag} = t^*] =$

$1/2^n$, Thus,

$$\Pr[\mathsf{Succ}_{\mathcal{A}}^{(\rho,h)}] = \frac{1}{2^n}. \tag{6.21}$$

Case II: There was a query with the nonce N^*: In this case, $\rho(N^*)$ was fixed, say to t when a query with nonce N^* was asked by $\mathcal{A}$. Thus the tag computed by the verification algorithm is

$$\mathsf{tag} = \mathsf{Poly}_h(M^*) \oplus h^{\lceil L^*/n \rceil} L^* \oplus t.$$

For $\mathcal{A}$ to be successful it is required that $\mathsf{tag} = t^*$. Thus, in this case, we have

$$\begin{aligned} \Pr[\mathsf{Succ}_{\mathcal{A}}^{(\rho,h)}] &= \Pr[\mathsf{tag} = t^*] && (6.22)\\ &= \Pr[\mathsf{Poly}_h(M^*) \oplus h^{\lceil L^*/n \rceil} L^* \oplus t \oplus t^* = 0]. && (6.23) \end{aligned}$$

The above probability is computed on the randomness of the choice of h. As,

$$p(h) = \mathsf{Poly}_h(M^*) \oplus h^{\lceil L^*/n \rceil} L^*] \oplus t \oplus t^*,$$

is a nonzero polynomial of degree $\lceil L^*/n \rceil$, hence there are at most $\lceil L^*/n \rceil$ many values of h which makes $p(h) = 0$. Thus, we have

$$\begin{aligned} \Pr[\mathsf{Succ}_{\mathcal{A}}^{(\rho,h)}] &= \Pr[\mathsf{tag} = t^*] && (6.24)\\ &\leq \frac{1}{2^n} \left\lceil \frac{L^*}{n} \right\rceil && (6.25)\\ &= \frac{1}{2^n} \left\lceil \frac{|M^*|}{n} \right\rceil. && (6.26) \end{aligned}$$

Now, combining cases I and II and using the union bound, we have

$$\Pr[\mathsf{Succ}_{\mathcal{A}}^{(\rho,h)}] \leq \frac{1}{2^n} \left(\frac{|M^*|}{n} + 1 \right) + \frac{1}{2^n} \tag{6.27}$$

Finally, using Equations (6.20) and (6.27), we get

$$\mathsf{Adv}^{\mathsf{Uauth}}_{\mathsf{XoRU}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{prf}}(\mathcal{B}) + \left(\frac{|M^*|+1}{n}\right)\frac{1}{2^n} + \frac{1}{2^n},$$

as desired. □

6.6 Final Remarks

We proposed a new type of MAC UdMAC which can be used to update the authentication tag of a message and compute the correct tag for the updated message only with access to the update. We defined the security of UdMACs and finally proposed two constructions. This class of MACs will be used in the next chapter to construct verifiable SSE schemes.

7

Fault-tolerant Verifiable DSSE

So far in this thesis, all the adversaries we've considered for an SSE are honest-but-curious. However, the server, which is typically viewed as the adversary in an SSE system, can potentially act maliciously by improperly storing user data or by providing incorrect responses to user queries in order to conserve storage and CPU resources. In addressing such adversarial behavior, SSE systems must incorporate a mechanism by which search results may be verified for correctness. An SSE equipped with a mechanism to verify the correctness of search queries is called a verifiable SSE.

It is not feasible for an honest client to keep track of the current state of the database stored on the server. As a result, a client may issue faulty updates, such as adding duplicate entries to the database or attempting to delete non-existent entries (see Definition 2.5.4). Since the database is encrypted, the server cannot assist the client in identifying such incorrect behavior. Informally, an SSE which inherently corrects such faulty updates is called a fault tolerant SSE.

Both verifiability and fault tolerance are essential characteristics required to safeguard the interests of clients. Recent work by [94] proposed the first fault-tolerant verifiable DSSE (FVDSSE) that achieves forward privacy.

In this work, we present the first generic construction for a DSSE scheme that is both forward and backward private, as well as verifiable and fault-tolerant. The construction by [94] provided the verification for the search results using an *authenticated encryption* (AE) scheme. For each update, in addition to the update token, they also outsource an additional AE tag, which they call an AE *proof*. When retrieving the

results, the server sends the AE proofs along with the results, and the client verifies the results using these AE proofs.

In an SSE system, the search result for a keyword w is the set $\mathsf{db}(w)$. An update to a keyword essentially modifies the set $\mathsf{db}(w)$, either by adding new identifiers or deleting some from the set. The previous schemes which ensure verifiability [24, 98, 48, 99] generally use an incremental multi-set, which is unable to provide proof in the presence of faulty updates. The work of [94] generates a proof for every update using an AE scheme and sends the update token and the proof/tag to the server. During the search, the server finds all the updates and corresponding tags and sends them to the client. The client verifies each update by invoking the AE verify algorithm.

In our proposed scheme, we see all the updates related to a keyword w as a message related to the keyword w which is getting updated with each update operation. Thus, using any updatable MAC, we can generate a tag which acts as proof for every update corresponding to a w. The specific syntax of a UdMAC allows us then to have a single tag of constant length for all the updates made for the keyword. This tag can be stored with the client, as most SSE schemes [23, 25, 24, 87, 49, 90, 38] allows a $\mathcal{O}(|\mathsf{W}|)$[1] storage at the client side. Please note that keeping the tag safe on the client side is not necessary for the purpose of security. We can also outsource the tag with the update query and ask the server to save the tag. In the proof, we'll provide the adversary with the tag and the state of the UdMAC algorithm for every keyword.

7.1 Generic Fault-tolerant Verifiable DSSE

In this section, we explain how to use an UdMAC scheme Ψ to transform any dynamic secure SSE Σ (see Section 2.3) which is also forward and backward private, into an equivalently secure dynamic forward and backward private verifiable SSE $\mathsf{v}\Sigma$ that supports faulty updates. We assume the existence of a forward and backward private secure DSSE scheme Σ, with leakage $\mathcal{L}_\Sigma$.

An updatable MAC is best described in a client-server framework. Where the

[1]Recall W is the set of distinct keyword present in the database.

client with the messages wants to store it in the server and then update the messages. While querying the client needs the guarantee that the server is sending the current state of the message and has not tampered with the message.

Recall the primary motivation behind a UdMAC is to avoid recomputing the MAC of an entire message when updates are made. This is particularly beneficial when we outsource the message and its tag (potentially to a third-party server). Without an updatable MAC, updating the message would necessitate downloading the entire message and its tag, verifying it, making updates, generating a new tag, and then outsourcing the updated message and tag together, which can be costly.

Notice that the MAC-update algorithm of UdMAC Ψ in Section 6.3 has three inputs. The secret key K, the message m, and a state σ, and outputs a tag t and updated state σ. Now, the state σ is specific to every message. Thus, the client has to maintain a state for every new message. Now, on query for a message, the server replies with the latest updated message along with its current tag.

A MAC algorithm dictates that only the key should be secret, and the adversary should have control over every other parameter in the forgery. We'll prove the security of our scheme in a model where the adversary controls every parameter of the MAC algorithm except the key. However, the client keeping the state of the MAC algorithm is a requirement of the model (for convenience), and has nothing to do with the security. This does not imply that we can outsource the state of the SSE from client to the server side, and still have correct verification. We believe that to outsource the state of the SSE we need to have some sort of verification on that as well. This needs further investigation.

7.1.1 A Generic FVDSSE Scheme $\mathsf{v}\Sigma$ Using Ψ and Σ

In the beginning, we fix an updatable-MAC scheme Ψ and a secure DSSE scheme Σ and demonstrate how to construct an FVDSSE scheme $\mathsf{v}\Sigma$. The setup process summarised in Figure 7-1, starts by initializing a key for the Ψ-MAC. A general practice for designing SSE is to assume an empty database at the setup and update it accordingly. We call the setup phase of Σ to get back a key K_Σ a state σ_C and

$\underline{\mathsf{Setup}(1^\lambda, \mathsf{DB})\textbf{:}}$
01. $K_{\mathsf{mac}} \leftarrow \Psi.\mathsf{KeyGen}\left(1^\lambda\right)$
02. $(K_\Sigma, \sigma_\mathsf{C}, \mathsf{EDB}) \leftarrow \Sigma.\mathsf{Setup}\left(1^\lambda, \perp\right)$
03. $\mathsf{UT}, \mathsf{ST} \leftarrow []$ \\ empty map
04. $\mathsf{K} \leftarrow (K_{\mathsf{mac}}, K_\Sigma)$
05. $\mathsf{State} \leftarrow (\sigma_\mathsf{C}, \mathsf{ST}, \mathsf{UT})$
06. Keep $(\mathsf{K}, \mathsf{State})$ to client
07. Send EDB to server

Figure 7-1: Setup phase of $\mathsf{v}\Sigma$.

an empty encrypted database EDB. We also initialize two empty maps UT and ST. These two maps are used to store the number of updates for a keyword and the state of the Ψ for each keyword, respectively.

$\underline{\mathsf{Update}(\mathsf{K}, \mathsf{State}, \mathsf{op}, (\mathsf{id}, w))\textbf{:}}$
Client Side:
01. **if** $\mathsf{UT}[w] = \perp$ **set** $\mathsf{UT}[w] \leftarrow 1$, **else** $\mathsf{UT}[w] \leftarrow \mathsf{UT}[w] + 1$
02. $\mathsf{ut}_w \leftarrow \mathsf{UT}[w]$
03. $\widetilde{\mathsf{id}} \leftarrow (\mathsf{ut}_w, \mathsf{op}, \mathsf{id})$
04. $(\sigma_\mathsf{C}, \mathsf{utk}) \leftarrow \Sigma.\mathsf{Update}_\mathsf{C}(K_\Sigma, \sigma_\mathsf{C}, \mathsf{add}, (\widetilde{\mathsf{id}}, w))$
05. **if** $\mathsf{ST}[w] = \perp$
06. $\quad (\sigma_{\mathsf{mac}}, \mathsf{tag}) \leftarrow \Psi.\mathsf{MacUpdate}(K_{\mathsf{mac}}, \mathsf{ST}[w], w)$
07. $\quad \mathsf{ST}[\mathsf{w}] \leftarrow (\sigma_{\mathsf{mac}}, \mathsf{tag})$
08. $(\sigma_{\mathsf{mac}}, \mathsf{tag}) \leftarrow \Psi.\mathsf{MacUpdate}(K_{\mathsf{mac}}, \mathsf{ST}[w], \widetilde{\mathsf{id}})$
09. $\mathsf{ST}[\mathsf{w}] \leftarrow (\sigma_{\mathsf{mac}}, \mathsf{tag})$
10. Send utk to server
Server Side:
20. Server updates EDB using utk

Figure 7-2: Update protocol of $\mathsf{v}\Sigma$.

In the update phase, for every update operation $(\mathsf{op}, \mathsf{id}, w)$, we fetch the update number ut_w from the map UT. If the keyword is being updated for the first time, we set the update number to 1. Otherwise, we increment the update number by 1. Then we construct a new $\widetilde{\mathsf{id}}$ as an encoding of $(\mathsf{ut}, \mathsf{op}, \mathsf{id})$. With this new $\widetilde{\mathsf{id}}$ we now call the base SSE scheme with update query $(\mathsf{add}, \widetilde{\mathsf{id}}, w)$.

If the keyword is being updated for the first time, we call the Ψ with an empty state and a $|\widetilde{\mathsf{id}}|$ bit encoding of the keyword w and save the state. That is, the first message for every keyword is the keyword itself. This is done to have a signature

of the keyword on every update on the keyword. We then update the message for the keyword with the current update $\widetilde{\mathsf{id}}$, and save the current state and tag for the keyword. Next, we call the base SSE scheme Σ with $(\mathsf{add}, \widetilde{\mathsf{id}}, w)$ to get back an update token utk. Finally, we send the update token (from SSE) to the server. The server updates the EDB with the utk. The process is summarised in Figure 7-2.

$\mathsf{Search}(\mathsf{K}, \mathsf{State}, w; \mathsf{stk}, \mathsf{EDB})$:
Client Side:
01. $\mathsf{stk} \leftarrow \Sigma.\mathsf{Search}_{\mathsf{C}}(K_{\Sigma}, \sigma_{\mathsf{C}}, w)$
02. Send stk to Server
Server Side:
10. **return** res to Client
Client Side:
20. $\mathsf{Result} \leftarrow \mathsf{Verify}(\mathsf{K}, \mathsf{State}, w, \mathsf{res})$
21. **return** Result

Figure 7-3: Search protocol of $\mathsf{v}\Sigma$.

During the search operation, the client calls the base SSE scheme with the search keyword w. The client provides the server with the search token it got from the base SSE scheme. The correctness of the base SSE dictates that the server is able to find all the updates related to the search keyword w with the search token. Upon receiving the search result the client runs the verify algorithm with the search result as input. The entire process of search is described in Figure 7-3.

The verify algorithm described in Figure 7-4 checks the search result output by the server and returns the correct search result upon verification else returns "reject". The algorithm first finds all $\widetilde{\mathsf{id}}_i$ of the form $i \| \mathsf{op}_i \| \mathsf{id}_i$ from the search result. Now from $i = 1$ to $\mathsf{UT}[w]$ in sequence, if $\mathsf{op}_i = \mathsf{add}$, then the algorithm adds the id_i to the result set, else if $\mathsf{op}_i = \mathsf{del}$ then the algorithm discards id_i from the result set. The algorithm also constructs a message m for verification. It initializes the message with a $|\widetilde{\mathsf{id}}|$ bit encoding of the keyword w, and append every $\widetilde{\mathsf{id}}_i$ for $1 \leq i \leq \mathsf{UT}[w]$ in sequence. Finally, the algorithm fetches the state of the keyword $\mathsf{ST}[w]$ and sends the state and the m it constructed to the $\Psi.\mathsf{MacVerify}$ algorithm. If the output of $\Psi.\mathsf{MacVerify}$ is "accept", then the algorithm returns the result set. Else, it returns "reject".

Verify($\mathsf{K}, \mathsf{State}, w, \mathsf{res}, \mathsf{tag}$):
00. $\mathsf{Result}, \mathsf{V} \leftarrow \emptyset$
01. $m \leftarrow w$
02. **for** $r \in \mathsf{res}$ **do**
03. extract $\widetilde{\mathsf{id}}$ from r and add it to V
04. $m \leftarrow m \| \widetilde{\mathsf{id}}$
05. **for** $i = 1$ to ut_w and $(i, \mathsf{op}, \mathsf{id}) \in \mathsf{V}$
06. **if** $\mathsf{op} = \mathsf{add}$
07. $\mathsf{Result} \leftarrow \mathsf{Result} \cup \{\mathsf{id}\}$
08. **else**
09. $\mathsf{Result} \leftarrow \mathsf{Result} \setminus \{\mathsf{id}\}$
10. $\beta \leftarrow \Psi.\mathsf{MacVerify}(K_{\mathsf{mac}}, m, \mathsf{ST}[w], \mathsf{tag})$
11. **if** $\beta = 1$
12. **return** Result
13. **else**
14. **return** Reject

Figure 7-4: Verify algorithm of $\mathsf{v}\Sigma$.

7.1.2 Fault-tolerance and Correctness of the Search Result

Notice that, upon verification, the Verify algorithm of $\mathsf{v}\Sigma$ returns a set of document identifiers matching the searched keyword w. Now, if $\mathsf{v}\Sigma$ is sound, that is, the server could not forge the search result, then the result returned by the server is the set of all ids updated for w. Recall, for an update query $(\mathsf{op}_i, \mathsf{id}_i, w)$ on keyword w, all updates to the $\mathsf{v}\Sigma$ is of the form $(\mathsf{add}, \widetilde{\mathsf{id}}_i, w)$, where $\widetilde{\mathsf{id}}_i$ is a λ bit encoding of the tuple $(i, \mathsf{op}_i, \mathsf{id}_i)$, and i is the number of updates performed on w till that point. This value i is kept at a map UT indexed by every keyword w in the database. After every update $\mathsf{UT}[w]$, which was initialized to 1 during the first update call on w, is incremented by 1. Thus, if the $\mathsf{v}\Sigma$ is sound, we get back all $\mathsf{UT}[w]$ updates on the keyword w. Now, the Verify algorithm of $\mathsf{v}\Sigma$ scans each $\widetilde{\mathsf{id}} = (i, \mathsf{op}_i, \mathsf{id}_i)$ in the result set res returned by the server in order of i. To compute the final result, the Verify algorithm initialises an empty set Result. For every $\mathsf{op}_i = \mathsf{add}$, it adds id to the result set (i.e., $\mathsf{Result} \leftarrow \mathsf{Result} \cup \{\mathsf{id}\}$). And for every $\mathsf{op}_i = \mathsf{del}$, it deletes id to the result set (i.e., $\mathsf{Result} \leftarrow \mathsf{Result} \setminus \{\mathsf{id}\}$). Thus, in the presence of a faulty update, that is add id while id is already in the database, or delete id where id is not in the database, the set operations performed by the algorithm Verify will ensure the correctness of the

search result. Details of the Verify algorithm can be found in Figure 7-4.

7.2 Comparisons and Actual Overheads

In this section, we compare the overheads of our scheme with some of the existing works in the literature. Table 7.1 provides a summary of selected studies focused on forward and backward private verifiable fault-tolerant DSSE. The table clearly shows that none of the existing works achieves verifiability along with backward privacy and fault tolerance. In this context, our construction is the first to accomplish all of these features.

Next, we compare the overheads of our scheme with those of existing constructions. The incremental multi-set hash proposed in [41] and adopted in [24, 98] involves multiplication over a DDH-hard group, which is a computationally expensive operation. Additionally, these schemes require storing the incremental tags on the server. The most efficient scheme presented in [94] uses an AE to generate and store proofs for every update. The construction proposed in [94] also introduces a generic method to integrate their approach with any forward-secure single-keyword SSE. However, it fails to achieve backward privacy. The instantiation in [94] utilizes the FAST protocol proposed in [87] as the single-keyword SSE. In this scheme, the AE proof generated for each update is outsourced and stored along with the update token of the FAST protocol on the server. During a query, the server retrieves all updates along with their proofs and returns them to the client, who then runs AE verification on each result to ensure correctness. The overhead of the most efficient FVDSSE scheme [94] is twice that of the base single-keyword SSE scheme, FAST, due to the storage and communication of the tags. Additionally, the scheme requires verification to be performed on each value returned by the server. The client-side storage is also doubled compared to what is required by the FAST protocol. In contrast, our protocol incurs no additional server-side storage and communication overhead as we save a fixed length tag for every update on the client side. This amount of storage is allowed in the DSSE scheme, achieving forward and backward privacy and sub-linear

Construction	Computation		Communication		Client Storage	FP	FT	BP
	Search	Update	Search	Update				
[64]	$\mathcal{O}(\lvert D\rvert)$	$\mathcal{O}(u\lvert D\rvert)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	N	N	N
GVS-Hash [24]	$\mathcal{O}(m+\log W)$	$\mathcal{O}(u\cdot\log W)$	$\mathcal{O}(m+\log W)$	$\mathcal{O}(\log W)$	$\mathcal{O}(1)$	Y	N	N
GVS-Acc-Pairing [24]	$\mathcal{O}(m)$	$\mathcal{O}(uW^{\epsilon})$	$\mathcal{O}(m+\log W)$	$\mathcal{O}(\log W)$	$\mathcal{O}(1)$	Y	N	N
GVS-Acc-RSA [24]	$\mathcal{O}(m+W^{\epsilon})$	$\mathcal{O}(u)$	$\mathcal{O}(m+\log W)$	$\mathcal{O}(\log W)$	$\mathcal{O}(1)$	Y	N	N
[98]	$\mathcal{O}(u)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(\lambda W)$	Y	N	N
Verifiable Linear SPS [24]	$\mathcal{O}(\alpha+\log N)$	$\mathcal{O}(u\cdot\log^2 N)$	$\mathcal{O}(m+\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\lambda\log N)$	Y	Y	N
Verifiable Sublinear SPS [24]	$\mathcal{O}(m\cdot\log^3 N)$	$\mathcal{O}(u\cdot\log^2 N)$	$\mathcal{O}(m+\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\lambda\log N)$	Y	Y	N
[94]	$\mathcal{O}(u)$	$\mathcal{O}(1)$	$\mathcal{O}(u)$	$\mathcal{O}(1)$	$\mathcal{O}(W\log\lvert D\rvert)$	Y	Y	N
vΣ [Section 7.1]	$\mathcal{O}(u)$	$\mathcal{O}(1)$	$\mathcal{O}(u)$	$\mathcal{O}(1)$	$\mathcal{O}(\lambda W)$	Y	Y	**Y**

Table 7.1: Comparison of our schemes with existing VDSSE schemes. FP, BP and FT stand for forward privacy, backward privacy, and fault-tolerant, respectively. 'Computation' and 'Communication' refer to the computational and communication complexity, respectively. W is the number of distinct keywords in the database, and $|D|$ denotes the number of documents in the database. N is the number of keyword-identifier pairs. $0 < \epsilon < 1$ is a fixed constant, and α denotes the number of times the queried keyword was historically added to the database. m denotes the number of matching entries for the searched keyword.

search [23, 25, 49, 38, 90]. Furthermore, storing the tag on the client side does not compromise security, as demonstrated in the security proof (Theorem 7.3.2) of our scheme, where the adversary is required to store and submit the tags at the time of query and forgery. We have added a row in Table 7.1 which reflects the overhead of our scheme in comparison to others. As our scheme can be instantiated with any DSSE scheme, to make this type of comparison we have to assume a particular construction. In this case, we assume the construction of [38] or [49].

It is clear from the description of vΣ any updatable MAC scheme satisfying the Definition 6.3 can be used as Ψ in our construction vΣ. So, we can use both our proposals ConCatU and XoRU as our Ψ in the construction of vΣ. Below we describe each extra overhead of vΣ while clubbed with each of the ConCatU or XoRU.

Overhead due to ConCatU. From the description of the efficiency of ConCatU in Section 6.4.1, it is clear that a single update call for a message m will require $\lceil(|m|+2)/n\rceil \approx \lceil|m|/n\rceil + 2$ block-cipher calls and one multiplication in $\mathbb{F}_p$, where $|m|$ is the size of the message and n is the block size. This also requires the ConCatU algorithm to save an intermediate value p, which is of size n at the client side. However, one might argue that ConCatU requires storing the list L of the size of every update made

to the message. The size of the list L is the order of the number of updates made for the keyword. So storing the list L will defeat the purpose of SSE.

We like to mention that, in the context of SSE, every update for a keyword w has a fixed size, that is, the size $\widetilde{\mathsf{id}}$. So the information of L is implicit in the context of $\mathsf{v}\Sigma$, and the algorithm $\mathsf{ConCatU}$ in the context of $\mathsf{v}\Sigma$ does not need to explicitly store the list L as its state. So the extra overhead of $\mathsf{v}\Sigma$ over the base SSE scheme Σ is the tag value for every keyword and the value of p, and nothing else. At this point, we would like to mention that most SSE schemes, which are forward and backward private [23, 25, 87, 38, 90], store the number update for every keyword, that is, the map UT at the client end. This can also be verified from Table 7.1. However, if the base SSE does not require to store UT then another extra overhead would be of order $\log |D|$.

Overhead due to XoRU. As discussed in the special case of Section 6.5.1, where XoRU is used only for concatenation, the message is always updated with a string of the form $0^{|M|}\|m$, where M is the previous message and m is the update which is to be concatenated with the current message. In scenarios where, $|m| = kn$, with k being a small constant, the update would require only $(2k+2)$ multiplications in $\mathbb{F}_{2^n}$ and 2 AES computations, provided that the value of h^{ℓ_1} is stored for every keyword. Please refer to Section 6.5.1 for a detailed analysis.

In the case of an SSE update, we update with only one (id, w) pair at a time. As a result, for each update the message length increases by only one n-bit block, i.e., the message m which is to be concatenated, is always n-bits long. Consequently, the Equation (6.19) boils down to the following:

$$(\ell_2 + 4)T_{\mathsf{add}} + 4T_{\mathsf{mult}} + 2T_{\mathsf{AES}},$$

where T_{add} and T_{mult} are the times taken for a multiplication and addition in $\mathbb{F}_{2^n}$ respectively and T_{AES} is the time taken for one call to AES. This amount of computation can be easily handled by a client in any reasonable computing platform.

7.3 Security

In the construction of the FVDSSE scheme $\mathsf{v}\Sigma$, we keep the tag generated for each update to the client itself. This is done to reduce the overhead of our scheme. However, this is not necessary, and the client may choose to outsource the tag generated for every update. Keeping or outsourcing the tag does not affect the security of the SSE. We prove both of our security requirements, i.e., adaptive security and soundness, by allowing the adversary to see the tag along with the update token. We consider that the SSE stores the tag in a similar fashion as it stores the id.

7.3.1 Adaptive Security of $\mathsf{v}\Sigma$

The adaptive security of SSE is established by capturing the maximum permissible leakage of a scheme and designing a simulator that, using these leakage functions, can generate a transcript indistinguishable from the one produced by the real-world execution of the protocol [60, 23, 38]. We assume that the leakage of the base SSE scheme Σ is $\mathcal{L}_\Sigma = (\mathcal{L}_{\Sigma,\mathsf{Setup}}, \mathcal{L}_{\Sigma,\mathsf{Search}}, \mathcal{L}_{\Sigma,\mathsf{Update}})$ as defined in Definition 5.2.2.

Our design, $\mathsf{v}\Sigma$, mirrors the setup phase of Σ exactly, resulting in identical setup leakage for both. In the update phase, along with the update token of Σ, we also send a tag for the update. The tag can be modelled as the output of a PRF following Proposition 6.4.1 and 6.5.2 in the real-world execution of our protocol. Therefore, our scheme has the same update leakage as the base SSE scheme Σ, as the tag value does not convey any information about the input of the update function. Finally, in the search protocol, we exactly follow the search protocol of the base SSE scheme. Thus, the leakage of protocol $\mathsf{v}\Sigma$ is $\mathcal{L}_{\mathsf{v}\Sigma} = \mathcal{L}_\Sigma$. Below, we formally prove the confidentiality and soundness of our scheme.

Theorem 7.3.1. *If Σ is a $\mathcal{L}_\Sigma$-adaptive secure forward and backward private DSSE scheme, then our protocol $\mathsf{v}\Sigma$ is also $\mathcal{L}_\Sigma$-adaptive secure forward and backward private DSSE.*

Proof. As Σ is an $\mathcal{L}_\Sigma$-adaptive secure forward and backward private DSSE scheme,

with the leakage function $\mathcal{L}_{\Sigma}$, it implies the existence of a simulator $\mathcal{S}$ that can, using the leakage function $\mathcal{L}_{\Sigma}$, simulate a transcript $\mathsf{T}^{\Sigma}_{\mathsf{sim}}$ indistinguishable from the real-world transcript $\mathsf{T}^{\Sigma}_{\mathsf{real}}$ of the protocol Σ.

In this proof, we will demonstrate that a simulator $\widetilde{\mathcal{S}}$ can efficiently generate a transcript $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{sim}}$ of our protocol using the transcript $\mathsf{T}^{\Sigma}_{\mathsf{sim}}$ produced by the simulator $\mathcal{S}$. Furthermore, we will show that any efficient distinguisher that can distinguish between the transcript $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{sim}}$ generated by $\widetilde{\mathcal{S}}$ and the real-world transcript of our protocol $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{real}}$ can always be used to construct a distinguisher for Σ, or a PRF used to model the UdMAC Ψ used in our construction.

Hence, if Σ is $\mathcal{L}_{\Sigma}$-adaptive secure, then our scheme is also $\mathcal{L}_{\Sigma}$-adaptive secure, as the leakage of Σ is exactly the same as the leakage of our scheme $\mathsf{v}\Sigma$. Therefore, the theorem. We demonstrate our proof using a sequence of games.

Game G_0: At the beginning of the protocol, the setup and every search query we follow the protocol $\mathsf{v}\Sigma$. For each update query $(\mathsf{op}, \mathsf{id}, w)$ made by the adversary $\mathcal{A}$, the challenger maintains a list UT that tracks the number of updates performed on the keyword w. For every update query $(\mathsf{op}, \mathsf{id}, w)$ to Σ, the challenger replaces id with $\widetilde{\mathsf{id}} = (\mathsf{UT}[w], \mathsf{op}, \mathsf{id})$ and sets the operation op to add, as shown in Figure 7-2. The Σ outputs an update token for $(\mathsf{add}, \widetilde{\mathsf{id}}, w)$.

Also, for every update query, the challenger generates a tag using the UdMAC Ψ, as described in lines 5 to 8 of Figure 7-2. Finally, the challenger sends the update token generated by Σ for input $(\mathsf{add}, \widetilde{\mathsf{id}}, w)$ and the tag generated by Ψ protocol. Thus,

$$\Pr[\mathsf{G}_0 = 1] \;=\; \Pr\left[\mathsf{SSEReal}^{\mathsf{v}\Sigma}_{\mathcal{A}}(\lambda, q) = 1\right].$$

Game G_1: In the next game, for every update query, we randomly sample a value from $\{0,1\}^{\tau}$ and replace the value of tag with this newly sampled value. Now for any adversary $\mathcal{A}$ that can distinguish between game G_0 and game G_1, we can construct an adversary $\mathcal{B}_1$ which can distinguish the output Ψ from a PRF, which contradicts the Proposition 6.4.1 and 6.5.2. Thus,

$$|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_2 = 1]| \;\leq\; \mathsf{Adv}^{\mathsf{prf}}_{\Psi,\mathcal{B}_1} \;\leq\; \mathsf{negl}(\lambda).$$

Recall that, Σ is a $\mathcal{L}_{\Sigma}$-adaptively secure SSE. Thus, for the real-world transcript produced by executing the protocol Σ, (denoted by $\mathsf{T}^{\Sigma}_{\mathsf{real}}$), there exists a simulator $\mathcal{S}$ that, with the assistance of $\mathcal{L}_{\Sigma}$, can generate a transcript $\mathsf{T}^{\Sigma}_{\mathsf{sim}}$ that is indistinguishable from $\mathsf{T}^{\Sigma}_{\mathsf{real}}$. Now, when all the updates of the transcript $\mathsf{T}^{\Sigma}_{\mathsf{real}}$ are appended with a randomly sampled value from $\{0,1\}^{\tau}$, the newly formed transcript is exactly the real-world execution of the protocol $\mathsf{v}\Sigma$. We denote this transcript as $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{real}}$.

Construction of $\widetilde{\mathcal{S}}$: Finally, we construct a simulator $\widetilde{\mathcal{S}}$ which can simulate an indistinguishable transcript from $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{real}}$ with the help of the indistinguishable transcript $\mathsf{T}^{\Sigma}_{\mathsf{sim}}$ output by the simulator $\mathcal{S}$. The construction of $\widetilde{\mathcal{S}}$ is the following.

In the setup phase, the challenger sends an empty database to the adversary, which is exactly how the simulator $\mathcal{S}$ will simulate it.

The search queries of $\mathsf{v}\Sigma$ are simulated exactly the same way as the search queries of $\mathsf{T}^{\Sigma}_{\mathsf{sim}}$, are simulated. This simulation is correct as in $\mathsf{v}\Sigma$ we use the search protocol of Σ unaltered.

For the update query, the simulator $\widetilde{\mathcal{S}}$ replace all the update tokens of $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{real}}$ (except the tag part) with the update token as simulated by $\mathcal{S}$. The tag is simulated with randomly sampled strings from $\{0,1\}^{\tau}$. The final transcript is exactly the transcript of $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{sim}}$. Formally,

$$\left|\Pr[\mathsf{G}_2 = 1] - \Pr\left[\mathsf{SSEIdeal}^{\mathsf{v}\Sigma}_{\mathcal{A},\widetilde{\mathcal{S}}}(\lambda) = 1\right]\right| \;\leq\; \mathsf{Adv}^{\Sigma}_{\mathcal{B}_2,\mathcal{S}} \;\leq\; \mathsf{negl}(\lambda).$$

Now any efficient distinguisher, distinguishing between the real-world transcript $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{real}}$ and the simulate transcript $\mathsf{T}^{\mathsf{v}\Sigma}_{\mathsf{sim}}$ of $\mathsf{v}\Sigma$ output by $\widetilde{\mathcal{S}}$, we can construct a distinguisher that distinguishes transcript simulated by $\mathcal{S}$ from the real-world execution of Σ. Thus,

$$\left|\Pr\left[\mathsf{SSEReal}^{\mathsf{v}\Sigma}_{\mathcal{A},\widetilde{\mathcal{S}}}(\lambda) = 1\right] - \Pr\left[\mathsf{SSEIdeal}^{\mathsf{v}\Sigma}_{\mathcal{A},\widetilde{\mathcal{S}}}(\lambda) = 1\right]\right| \;\leq\; \mathsf{negl}(\lambda).$$

□

7.3.2 Forward and Backward Privacy of $\mathsf{v}\Sigma$

A DSSE scheme is said to be forward and backward private if the leakage of that scheme is bounded by some well defined leakage functions defined in Section 2.5.5. Now, in Theorem 7.3.1 we proved that if the base SSE scheme Σ used in our $\mathsf{v}\Sigma$ construction has leakage $\mathcal{L}_\Sigma$, then our FVDSSE scheme $\mathsf{v}\Sigma$ also has the same leakage $\mathcal{L}_\Sigma$. As the base SSE scheme Σ is forward and backward private, the $\mathcal{L}_\Sigma$ is subsumed by the leakage defined in the forward and backward privacy Definition 2.5.5 and 2.5.6 respectively. Thus leakage of $\mathsf{v}\Sigma$ is also subsumed by the leakage defined in the Definition 2.5.5 and 2.5.6. So, by Definition 2.5.5 and 2.5.6, our construction $\mathsf{v}\Sigma$ also achieves forward privacy and the same level of backward privacy as achieved by the base SSE scheme Σ.

7.3.3 Soundness of $\mathsf{v}\Sigma$

While arguing about soundness of $\mathsf{v}\Sigma$ we let the adversary play the soundness game of DSSE as defined in Definition 2.5.3. We prove in the following theorem that if $\mathsf{v}\Sigma$ is instantiated with a correct SSE scheme Σ as in Definition 2.5.2, then given any adversary $\mathcal{A}$ breaking the soundness of $\mathsf{v}\Sigma$ as in Definition 2.5.3, we can construct another adversary $\mathcal{B}$ which can either break the authenticity of UdMAC Ψ or can break the correctness of Σ.

The adversary $\mathcal{A}$ initially submits a database and receives the encrypted database. In the subsequent phase, $\mathcal{A}$ makes a search and update query of its choice. Adversary $\mathcal{B}$ acts as a challenger for $\mathcal{A}$ and responds to the query of $\mathcal{A}$ by running an SSE instance on its own. For the tag, it sends along with the update query, $\mathcal{B}$ uses its challenger to compute the tag. Finally, any successful forgery by $\mathcal{A}$ according to Definition 2.5.3 we show that $\mathcal{B}$ can construct a successful forgery for Ψ.

Theorem 7.3.2. *If Σ is a correct DSSE scheme according to Definition 2.5.2, then for any* ppt *adversary $\mathcal{A}$ which can break the soundness of $\mathsf{v}\Sigma$ as in Definition 2.5.3, there exists another* ppt *adversary $\mathcal{B}$ which will break the security of UdMAC Ψ as defined in Section 6.3.1.*

Proof. We establish the soundness of our scheme $\mathsf{v}\Sigma$ under the assumption that Σ is always correct. This implies that given the latest search token stk for a keyword w, the server can always accurately retrieve all updates made to EDB for keyword w up to that point. Failure to satisfy this condition would result in the soundness of $\mathsf{v}\Sigma$, reducing to the correctness of Σ.

Let's assume a ppt adversary $\mathcal{A}$, which makes polynomial many queries to the SSE $\mathsf{v}\Sigma$, in the soundness game of Definition 2.5.3 and wins. We then construct another UdMAC adversary $\mathcal{B}$ that uses $\mathcal{A}$ as a subroutine to break the security of Ψ as described in Section 6.3.1. We follow the security definition of 2.5.3 and allow the adversary to manipulate the search results arbitrarily. Adversary $\mathcal{B}$ acts as the challenger of $\mathcal{A}$ and plays the soundness game defined in Definition 2.5.3. $\mathcal{B}$ runs an SSE instance on its own and only uses its challenger for Ψ to compute the tags. $\mathcal{B}$ runs $\Sigma.\mathsf{Setup}$ to obtain a key, a state, and an encrypted database EDB for the SSE. $\mathcal{B}$ then sends EDB to $\mathcal{A}$. Here, we assume the common practice of outsourcing an empty database during setup and then updating it. In cases where the initial database is not empty, we can still outsource an empty database and update it subsequently.

Now, for every update query $(\mathsf{op}, \mathsf{id}, w)$ that adversary $\mathcal{A}$ makes, adversary $\mathcal{B}$ runs $\Sigma.\mathsf{Update}_{\mathsf{C}}$ and returns the utk to adversary $\mathcal{A}$ following soundness game of Figure 2-2. Along with the utk, adversary $\mathcal{B}$, following steps 5 to 8 of the update algorithm of $\mathsf{v}\Sigma$ in Figure 7-2, generates the tag for the update query. To generate the tag, adversary $\mathcal{B}$, queries its challenger for the UdMAC Ψ. To do this, adversary $\mathcal{B}$ maintains the state ST and UT described in Figure 7-2. The adversary $\mathcal{B}$ sends the updated tag received from its challenger along with the update token utk. For a search query on keyword w from adversary $\mathcal{A}$, adversary $\mathcal{B}$ runs the $\Sigma.\mathsf{Search}$ protocol and returns the search token stk to $\mathcal{A}$.

Finally, the adversary stops by outputting a forgery for any search query of its choice. However, in response to any search query, it must return the correct number of search results corresponding to the number of updates made for the keyword. This can easily be checked by the adversary $\mathcal{B}$ using the map UT that it stores. Sending an equal number of search results is necessary as the correctness of Ψ can verify any

partial search result. In this security game, the adversary submits a search result along with a tag as its forgery.

For every update query, adversary $\mathcal{B}$ also maintains its own state D, initialized to empty. The state D is a list of sets for each keyword w. For every update operation $(\mathsf{op}, \mathsf{id}, w)$, adversary $\mathcal{B}$ performs the following steps:

- If $\mathsf{op} = \mathsf{add}$, $\mathsf{D}[w] \leftarrow \mathsf{D}[w] \cup \{\mathsf{id}\}$.
- If $\mathsf{op} = \mathsf{del}$, $\mathsf{D}[w] \leftarrow \mathsf{D}[w] \setminus \{\mathsf{id}\}$.

Finally, for a forgery attempt by $\mathcal{A}$ for a search query on a keyword w, $\mathcal{B}$ constructs it forgery as follows. Let the forgery of $\mathcal{A}$ be $\mathsf{res} = (L, t)$ consists of two components: (i) the list of all updates L (or an encrypted list, depending on the description of Σ) made for the keyword w up to that point, and (ii) a tag t. If $|L| \neq \mathsf{UT}[w]$, then res is not a valid forgery by $\mathcal{A}$, and $\mathcal{B}$ aborts. Otherwise, the entries of L (or the decrypted version) are of the form $(i, \mathsf{op}, \mathsf{id})$, where i is the update sequence number, op is the operation corresponding to that update, and id is the identifier involved in that update.

Now the adversary $\mathcal{B}$ constructs its forgery as in Figure 7-5. Notice, if $\widetilde{\mathsf{D}} = \mathsf{D}[w]$ in

Forgery of $\mathcal{B}$:

01. $m \leftarrow w$
02. $\widetilde{\mathsf{D}} \leftarrow \emptyset$
03. **for** $i = 1$ to $|L|$, **and** $\ell = i \| \mathsf{op} \| \mathsf{id} \in L$ **do**
04. $\quad m \leftarrow m \| \ell$
05. $\quad$ **if** $\mathsf{op} = \mathsf{add}$
06. $\quad\quad \widetilde{\mathsf{D}} \leftarrow \widetilde{\mathsf{D}} \cup \{\mathsf{id}\}$
07. $\quad$ **else if** $\mathsf{op} = \mathsf{del}$
08. $\quad\quad \widetilde{\mathsf{D}} \leftarrow \widetilde{\mathsf{D}} \setminus \{\mathsf{id}\}$
09. $\quad$ **end if**
10. **end for**
11. **if** $\widetilde{\mathsf{D}} = \mathsf{D}[w]$
12. $\quad$ abort
13. **return** $(m, \mathsf{ST}[w], t)$

Figure 7-5: Forger game of FVDSSE.

Line No. 11 of Figure 7-5, then res is not a valid forgery attempt by $\mathcal{A}$, and $\mathcal{B}$ aborts.

If the forgery submitted by $\mathcal{A}$ is a valid forgery then adversary $\mathcal{B}$ constructs its forgery as $(m, \mathsf{ST}[w], t)$ and submits to its challenger of Ψ. Notice that, if $\mathcal{A}$ submits a valid forgery, that is $\widetilde{\mathsf{D}} \neq \mathsf{D}[w]$ then the message constructed in Line Number 03 to 10 of Figure 7-5 has never been queried by $\mathcal{B}$ to its challenger. Thus, $(m, \mathsf{ST}[w], t)$ is a valid forgery attempt by $\mathcal{B}$. So, $\mathcal{A}$ breaking the soundness of $\mathsf{v}\Sigma$ implies, $\mathcal{B}$ certainly breaks the Ψ-MAC.

At this point, we allow the underlying SSE to be correct except with negligible probability. By Σ_{corr} we denote the event that the Σ is correct. So,

$$\begin{aligned}
\Pr[\mathcal{A} \text{ success}] &= \Pr\left[\mathcal{B} \text{ wins} \mid \Sigma_{\mathsf{corr}}\right] \Pr\left[\Sigma_{\mathsf{corr}}\right] + \Pr\left[\mathcal{B} \text{ wins} \mid \overline{\Sigma_{\mathsf{corr}}}\right] \Pr\left[\overline{\Sigma_{\mathsf{corr}}}\right] \\
&\leq \Pr\left[\mathcal{B} \text{ wins} \mid \Sigma_{\mathsf{corr}}\right] + \Pr\left[\overline{\Sigma_{\mathsf{corr}}}\right] \\
\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Sound},\Sigma} &\leq \mathsf{Adv}_{\mathcal{B}}^{\Psi} + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Corr},\Sigma}.
\end{aligned}$$

□

7.4 Final Remark

In this chapter, we addressed the open problem of designing a fault-tolerant verifiable SSE, which is both forward and backward private. We provided the first generic construction of FVDSSE using a generic DSSE which is only secure against an honest-but-curious adversary and our newly proposed updatable-MAC. Our construction is generic and requires no extra communication and storage on the server side, with the expense of one tag per keyword stored on the client side. Compared to other state-of-the art SSE which store and communicate one tag per update on the server side. A storage of $\mathcal{O}(|\mathsf{W}|)$ is considered acceptable in the SSE literature. In fact, almost all DSSE schemes keep a client state of $\mathcal{O}(|\mathsf{W}|)$.

8 Conclusion

In this thesis, we explored the security and efficiency of SSE under different adversarial settings and for various classes of queries.

To recall, we started this thesis with a four-way goal of (i) storing the inverted index in a succinct manner to reduce overheads of SSE, (ii) possibly supporting richer queries while maintaining efficiency, and (iii) & (iv) provide correct search result in presence of a malicious server and a client producing faulty update. To this end, we conclude that we have addressed all of the above objectives in a satisfactory way.

To address the first problem, we describe a novel way to make SSE schemes space efficient. Our scheme converts a given database into a different representation, which, on average, results in significant space savings. The smaller representation also results in reduced search time and response sizes. Our scheme depends on representing the set $\mathsf{db}(w)$ using binary trees where the leaves are labeled. This representation has interesting combinatorial properties, which we explore in detail. Our experiments show that our representation results in smaller index sizes with minimal extra overhead. Our scheme can be used with any secure single keyword SSE, and our scheme, being just a pre-processing step, retains the security of the base SSE.

Regarding our second goal, we have proposed a generic DSSE scheme that allows conjunctive queries while being both forward and backward private. Our scheme is specifically crafted to protect *non-modifiable* documents that, once uploaded, their keywords remain unaltered. This is a common characteristic in various applications, including biometric databases, digital archives, legal documents, medical records,

etc. Our construction does allow the keyword set associated with a document to be modified, but with some extra overhead. With efficiency closest to OXT (among known dynamic schemes) and by ensuring both forward and backward privacy, our proposed construction offers an ideal solution for preserving the privacy of dynamic datasets while facilitating conjunctive queries.

To achieve our third and fourth objective, we first introduce a novel message authentication scheme called the updatable MAC. Unlike traditional MAC methods, where updating a message necessitates recomputing the tag for the entire updated message, our updatable MAC system maintains a small state for each message. When a message is modified, our scheme recalculates the tag using only the stored state and the previous tag, eliminating the need to recompute the tag for the entire updated message. We establish specific security criteria for these MACs and propose two distinct MAC designs tailored for two different types of update functions. Out constructions support, updating the message through concatenation and XOR-difference.

Finally, we focus on designing a secure SSE scheme that is resilient against a malicious server, starting from an SSE scheme that is only secure against an honest-but-curious server. By using the updatable MAC introduced in this thesis, we convert a generic single keyword SSE scheme–originally forward and backward private against honest-but-curious adversaries–into a scheme that also offers security against malicious servers while maintaining forward and backward privacy. This approach is the first of its kind. We also achieve it without increasing communication or storage on the server's end from the base SSE construction. We only store a constant size state for every keyword at the client's end, which is well-accepted in SSE literature. On the contrary, state-of-the-art SSE only provides forward privacy and needs to store and communicate one tag per update.

Throughout our thesis, we provide generic constructions that use an SSE scheme in a black-box manner to create new SSE systems with desired security or performance requirements. This increases the acceptability of these works as these works can be integrated into existing systems without needing to modify the whole structure.

Future Directions

We list some aspects of our work which we plan to explore in the near future:

1. Our current study on cover based SSE involves only single keyword select queries. It seems that with some small modifications, it may be possible to equip the scheme to handle range queries.

2. We compute an estimate of the average size of a cover in the case of pure covers. Our estimate is quite accurate, as demonstrated by the experiments, but we failed to obtain a semantically useful analytical expression of the estimate. It would be nice to try to estimate the average size in some alternative way to obtain a more meaningful expression.

3. We would like to see the performance of our cover-based scheme in a real environment, which would require careful implementation of our scheme paired with a base SSE in a real cloud environment.

4. An attractive property of the cover based SSE scheme is that given the response of a query produced by our scheme, it is not possible to know the number of documents that match the query directly. This hints that our scheme is volume hiding to some extent. But, a thorough investigation of this property is required. We think that by using some additional randomization, it may be possible to convert our scheme into a volume hiding scheme against a large class of powerful adversaries.

5. The recent C-SSE scheme of [78] is similar to [29]. However, [78] pre-computes all possible 2-conjunctions to facilitate conjunctive search. This is difficult to achieve in a dynamic setting where the database is not known beforehand. It will be interesting to use [78] instead of OXT for Γ without a significant loss in efficiency.

6. The OXT framework has been extended to support multi-client settings in a modular black-box fashion [57]. Our construction also uses OXT in a black-box

manner without altering its functionality. Therefore, our generic scheme may be extended to support multi-clients. We leave this as another future work.

7. We provide constructions for updatable macs for two kinds of updates, namely, concatenation and xor difference. An interesting theoretical question could be to ask which kinds of updates can be handled by uddatable MACs. Also it would be interesting to design schemes for other update functions.

8. In the context of our work on verifiable SSE, a natural question is: Is it possible to use updatable MAC to construct a verifiable SSE, which supports more complex queries than a single keyword search? We wish to take this up in near future.

Bibliography

[1] Mohamed Ahmed Abdelraheem, Tobias Andersson, Christian Gehrmann, and Cornelius Glackin. Practical attacks on relational databases protected via searchable encryption. In Liqun Chen, Mark Manulis, and Steve A. Schneider, editors, *Information Security - 21st International Conference, ISC 2018, Guildford, UK, September 9-12, 2018, Proceedings*, volume 11060 of *Lecture Notes in Computer Science*, pages 171–191. Springer, 2018.

[2] Mohamed Ahmed Abdelraheem, Christian Gehrmann, Malin Lindström, and Christian Nordahl. Executing boolean queries on an encrypted bitmap index. In *Proceedings of the 2016 ACM on Cloud Computing Security Workshop*, CCSW '16, page 11–22. Association for Computing Machinery, 2016.

[3] Mohamed Ahmed Abdelraheem, Theo Henault Sanjay Bhattacherjee, and Avishek Majumder. Conjunctive dynamic searchable symmetric encryption – a generic framework. *The IACR Communications in Cryptology (CiC) (submitted).*

[4] Shweta Agrawal, Subhashis Banerjee, and Subodh Sharma. Privacy and security of aadhaar: A computer science perspective. *Economic and Political Weekly*, 52(37):93–102, 2017.

[5] Shweta Agrawal, Sanjay Bhattacherjee, Duong Hieu Phan, Damien Stehlé, and Shota Yamada. Efficient public trace and revoke from standard assumptions: Extended abstract. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2277–2293. Association for Computing Machinery, 2017.

[6] Ghous Amjad, Seny Kamara, and Tarik Moataz. Injection-secure structured and searchable symmetric encryption. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security,*

Guangzhou, China, December 4-8, 2023, Proceedings, Part VI, volume 14443 of *Lecture Notes in Computer Science*, pages 232–262. Springer, 2023.

[7] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *Proc. Priv. Enhancing Technol.*, 2023(1):417–436, 2023.

[8] Nishant Anand. New principles for governing aadhaar: Improving access and inclusion, privacy, security, and identity management. *Journal of Science Policy & Governance*, 18(01):1–14, 2021.

[9] John Christopher Anderson. Transmitting legal documents over the internet: How to protect your client and yourself. *Rutgers Computer & Tech. LJ*, 27:1, 2001.

[10] Vivek Arte, Mihir Bellare, and Louiza Khati. Incremental cryptography revisited: Prfs, nonces and modular design. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology - INDOCRYPT 2020 - 21st International Conference on Cryptology in India, Bangalore, India, December 13-16, 2020, Proceedings*, volume 12578 of *Lecture Notes in Computer Science*, pages 576–598. Springer, 2020.

[11] Arnab Bag, Debadrita Talapatra, Ayushi Rastogi, Sikhar Patranabis, and Debdeep Mukhopadhyay. Two-in-one-sse: Fast, scalable and storage-efficient searchable symmetric encryption for conjunctive and disjunctive boolean queries. *Proc. Priv. Enhancing Technol.*, 2023(1):115–139, 2023.

[12] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1994.

[13] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 45–56. ACM, 1995.

[14] Sanjay Bhattacherjee and Palash Sarkar. Complete tree subset difference broadcast encryption scheme and its analysis. *Des. Codes Cryptogr.*, 66(1-3):335–362, 2013.

[15] John Black and Phillip Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2000.

[16] John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 384–397. Springer, 2002.

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[18] Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 559–589. Springer, 2020.

[19] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 410–428. Springer, 2013.

[20] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. (this reference is still in the form of an incomplete draft.). 2023.

[21] Joppe W. Bos, Kristin Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics*, 50:234–243, 2014. Special Issue on Informatics Methods in Medical Privacy.

[22] Christoph Bösch, Richard Brinkman, Pieter H. Hartel, and Willem Jonker. Conjunctive wildcard search over encrypted data. In Willem Jonker and Milan Petkovic, editors, *Secure Data Management - 8th VLDB Workshop, SDM 2011, Seattle, WA, USA, September 2, 2011, Proceedings*, volume 6933 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2011.

[23] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1143–1154. Association for Computing Machinery, 2016.

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

[25] 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, CCS '17, page 1465–1482. Association for Computing Machinery, 2017.

[26] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, 1977.

[27] 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*, CCS '15, page 668–679. Association for Computing Machinery, 2015.

[28] 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 *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[29] 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 Berlin Heidelberg, 2013.

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

[31] Debrup Chakraborty, Avishek Majumder, and Subhabrata Samajder. Making searchable symmetric encryption schemes smaller and faster. *Int. J. Inf. Sec.*, 24(1):10, 2025.

[32] Debrup Chakraborty and Palash Sarkar. On modes of operations of a block cipher for authentication and authenticated encryption. *Cryptogr. Commun.*, 8(4):455–511, 2016.

[33] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2425–2442. USENIX Association, 2022.

[34] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455, 2005.

[35] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010*, pages 577–594, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[36] Melissa Chase and Emily Shen. Substring-searchable symmetric encryption. *Proc. Priv. Enhancing Technol.*, 2015(2):263–281, 2015.

[37] Sanjit Chatterjee, Manish Kesarwani, Jayam Modi, Sayantan Mukherjee, Shravan Kumar Parshuram Puria, and Akash Shah. Secure and efficient wildcard search over encrypted data. *Int. J. Inf. Sec.*, 20(2):199–244, 2021.

[38] Sanjit Chatterjee, Shravan Kumar Parshuram Puria, and Akash Shah. Efficient backward private searchable encryption. *J. Comput. Secur.*, 28(2):229–267, 2020.

[39] Valerio Cini, Sebastian Ramacher, Daniel Slamanig, Christoph Striecks, and Erkan Tairi. Updatable signatures and message authentication codes. In Juan A. Garay, editor, *Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part I*, volume 12710 of *Lecture Notes in Computer Science*, pages 691–723. Springer, 2021.

[40] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003*, pages 188–207, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[41] Dwaine E. Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*, volume 2894 of *Lecture Notes in Computer Science*, pages 188–207. Springer, 2003.

[42] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 79–88, New York, NY, USA, 2006. Association for Computing Machinery.

[43] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[44] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. 2021.

[45] Enron Corp and Cohen, W. W. Enron email dataset. United States Federal Energy Regulatory Commissioniler, comp, 2015.

[46] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 98–129. Springer, 2017.

[47] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, volume 9327 of *Lecture Notes in Computer Science*, pages 123–145. Springer, 2015.

[48] Xinrui Ge, Jia Yu, Hanlin Zhang, Chengyu Hu, Zengpeng Li, Zhan Qin, and Rong Hao. Towards achieving keyword search over dynamic encrypted cloud data with symmetric-key based verification. *IEEE Transactions on Dependable and Secure Computing*, 18(1):490–504, 2021.

[49] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1038–1055. Association for Computing Machinery, 2018.

[50] Eu-Jin Goh. Secure indexes. *IACR Cryptol. ePrint Arch.*, page 216, 2003.

[51] Cheng Guo, Wenfeng Li, Xinyu Tang, Kim-Kwang Raymond Choo, and Yining Liu. Forward private verifiable dynamic searchable symmetric encryption with efficient conjunctive query. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[52] Yu Guo, Chen Zhang, and Xiaohua Jia. Verifiable and forward-secure encrypted search using blockchain techniques. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–7, 2020.

[53] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. Practical and secure substring search. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 163–176. ACM, 2018.

[54] Changhui Hu and Lidong Han. Efficient wildcard search over encrypted data. *Int. J. Inf. Sec.*, 15(5):539–547, 2016.

[55] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *NDSS*, volume 20, page 12. Citeseer, 2012.

[56] Tetsu Iwata and Kaoru Kurosawa. OMAC: one-key CBC MAC. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153. Springer, 2003.

[57] 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*, CCS '13, page 875–888, New York, NY, USA, 2013. Association for Computing Machinery.

[58] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology – EUROCRYPT 2017*, pages 94–124. Springer International Publishing, 2017.

[59] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 183–213. Springer, 2019.

[60] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 965–976, New York, NY, USA, 2012. Association for Computing Machinery.

[61] Els J Kindt. *Privacy and data protection issues of biometric applications*, volume 1. Springer, 2016.

[62] Kaoru Kurosawa and Tetsu Iwata. TMAC: two-key CBC MAC. In Marc Joye, editor, *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, volume 2612 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2003.

[63] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 285–298, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[64] Kaoru Kurosawa and Yasuhiro Ohtaki. How to update documents verifiably in searchable symmetric encryption. In Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab, editors, *Cryptology and Network Security*, pages 309–328, Cham, 2013. Springer International Publishing.

[65] Kaoru Kurosawa, Keisuke Sasaki, Kiyohiko Ohta, and Kazuki Yoneyama. Uc-secure dynamic searchable symmetric encryption scheme. In Kazuto Ogawa and Katsunari Yoshioka, editors, *Advances in Information and Computer Security*, pages 73–90, Cham, 2016. Springer International Publishing.

[66] 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*, CCS '18, page 745–762. Association for Computing Machinery, 2018.

[67] Feng Li, Jianfeng Ma, Yinbin Miao, Pengfei Wu, and Xiangfu Song. Beyond volume pattern: Storage-efficient boolean searchable symmetric encryption with suppressed leakage. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part I*, volume 14344 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2023.

[68] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[69] Jinlu Liu, Bo Zhao, Jing Qin, Xi Zhang, and Jixin Ma. Multi-keyword ranked searchable encryption with the wildcard keyword for data sharing in cloud computing. *Comput. J.*, 66(1):184–196, 2023.

[70] Judith A Markowitz. Voice biometrics. *Communications of the ACM*, 43(9):66–73, 2000.

[71] Meixia Miao, Yunling Wang, Jianfeng Wang, and Xinyi Huang. Verifiable database supporting keyword searches with forward security. volume 77, page 103491, 2021.

[72] Tarik Moataz, Indrajit Ray, Indrakshi Ray, Abdullatif Shikfa, Frédéric Cuppens, and Nora Cuppens. Substring search over encrypted data. *J. Comput. Secur.*, 26(1):1–30, 2018.

[73] Tarik Moataz and Abdullatif Shikfa. Boolean symmetric searchable encryption. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 265–276. ACM, 2013.

[74] Mridul Nandi. Fast and secure CBC-Type MAC algorithms. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 375–393. Springer, 2009.

[75] Dalit Naor, Moni Naor, and Jeff Lotspiech. Revocation and tracing schemes for stateless receivers. In *Advances in Cryptology — CRYPTO 2001*, pages 41–62, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[76] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, page 191–198, New York, NY, USA, 2007. Association for Computing Machinery.

[77] Our Code. https://github.com/anonCod/CD-SSE.

[78] Sarvar Patel, Giuseppe Persiano, Joon Young Seo, and Kevin Yeo. Efficient boolean search over encrypted data with reduced leakage. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 577–607. Springer, 2021.

[79] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 79–93, 2019.

[80] Sikhar Patranabis and Debdeep Mukhopadhyay. Forward and backward private conjunctive searchable symmetric encryption. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[81] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, pages 16–31, 2004.

[82] Moe Sabry and Reza Samavi. ArchiveSafe LT: Secure long-term archiving system. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 936–948, New York, NY, USA, 2022. Association for Computing Machinery.

[83] Moe Sabry, Reza Samavi, and Douglas Stebila. Archivesafe: Mass-leakage-resistant storage from proof-of-work. In Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomarti, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 89–107, Cham, 2020. Springer International Publishing.

[84] James Scheibner, Jean Louis Raisaro, Juan Ramón Troncoso-Pastoriza, Marcello Ienca, Jacques Fellay, Effy Vayena, and Jean-Pierre Hubaux. Revolutionizing medical data sharing using advanced privacy-enhancing technologies: Technical, legal, and ethical synthesis. *J Med Internet Res*, 23(2):e25120, Feb 2021.

[85] Saeed Sedghi, Peter van Liesdonk, Svetla Nikova, Pieter H. Hartel, and Willem Jonker. Searching keywords with wildcards on encrypted data. In Juan A. Garay and Roberto De Prisco, editors, *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, volume 6280 of *Lecture Notes in Computer Science*, pages 138–153. Springer, 2010.

[86] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55, 2000.

[87] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized i/o efficiency. *IEEE Transactions on Dependable and Secure Computing*, 17(5):912–927, 2020.

[88] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. Internet Society, 2014.

[89] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 763–780. Association for Computing Machinery, 2018.

[90] Shifeng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph K. Liu, Surya Nepal, and Dawu Gu. Practical non-interactive searchable encryption with forward and backward privacy. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[91] Na Wang, Junsong Fu, Bharat K. Bhargava, and Jiwen Zeng. Efficient retrieval over documents encrypted by attributes in cloud computing. *IEEE Transactions on Information Forensics and Security*, 13(10):2653–2667, 2018.

[92] Mark N Wegman and J Lawrence Carter. New classes and applications of hash functions. In *20th Annual Symposium on Foundations of Computer Science (SFCS 1979)*, pages 175–182. IEEE, 1979.

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

[94] Dandan Yuan, Shujie Cui, and Giovanni Russello. We can make mistakes: Fault-tolerant forward private verifiable dynamic searchable symmetric encryption. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*, pages 587–605. IEEE, 2022.

[95] Dandan Yuan, Cong Zuo, Shujie Cui, and Giovanni Russello. Result-pattern-hiding conjunctive searchable symmetric encryption with forward and backward privacy. *Proc. Priv. Enhancing Technol.*, 2023(2):40–58, 2023.

[96] Xi Zhang, Ye Su, Zhongkai Wei, Wenting Shen, and Jing Qin. Efficient wildcard searchable symmetric encryption with forward and backward security. In Jaideep Vaidya, Moncef Gabbouj, and Jin Li, editors, *Artificial Intelligence Security and Privacy - First International Conference on Artificial Intelligence Security and Privacy, AIS&P 2023, Guangzhou, China, December 3-5, 2023, Proceedings, Part I*, volume 14509 of *Lecture Notes in Computer Science*, pages 342–357. Springer, 2023.

[97] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 707–720. USENIX Association, 2016.

[98] Zhongjun Zhang, Jianfeng Wang, Yunling Wang, Yaping Su, and Xiaofeng Chen. Towards efficient verifiable forward secure searchable symmetric encryption. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 304–321, Cham, 2019. Springer International Publishing.

[99] Jie Zhu, Qi Li, Cong Wang, Xingliang Yuan, Qian Wang, and Kui Ren. Enabling generic, verifiable, and secure data search in cloud services. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1721–1735, 2018.

[100] Cong Zuo, Shangqi Lai, Xingliang Yuan, Joseph K. Liu, Jun Shao, and Huaxiong Wang. Searchable encryption for conjunctive queries with extended forward and backward privacy. *IACR Cryptol. ePrint Arch.*, page 1585, 2021.

[101] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption with forward and stronger backward privacy. In *Computer Security – ESORICS 2019*, pages 283–303. Springer International Publishing, 2019.

[102] Cong Zuo, Shifeng Sun, Joseph K. Liu, Jun Shao, Josef Pieprzyk, and Guiyi Wei. Forward and backward private dynamic searchable symmetric encryption for conjunctive queries. *IACR Cryptol. ePrint Arch.*, page 1357, 2020.