Queryable Encryption for Outsourced Dynamic Data

A thesis submitted to Indian Statistical Institute
in partial fulfillment of the thesis requirements for the degree of
Doctor of Philosophy in Computer Science

Author:
Laltu Sardar

Cryptology and Security Research Unit
Indian Statistical Institute
203 B. T. Road, Kolkata, West Bengal, India - 700108

Supervisors:
Assoc. Prof. Sushmita Ruj
&
Prof. Bimal Kumar Roy

August 2021

Dedicated to

My Parents and Sisters

DECLARATION OF AUTHORSHIP

I, **Laltu Sardar**, a student of Cryptology and Security Research Unit, of the Ph.D. program of Indian Statistical Institute, Kolkata, hereby declare that the investigations presented in this thesis are based on my own works and, to the best of my knowledge, the materials contained in this thesis have not previously been published or written by any other person, nor it has been submitted as a whole or as a part for any degree/diploma or any other academic award anywhere before.

Laltu Sardar
23/08/2021

Laltu Sardar
Cryptology and Security Research Unit,
Indian Statistical Institute, Kolkata
203 Barrackpore Trunk Road,
Kolkata, West Bengal, India - 700108

CERTIFICATE FROM SUPERVISORS

This is to certify that the work contained in the thesis entitled **"Queryable Encryption for outsourced dynamic Data"**, submitted by **Laltu Sardar** for the award of the degree of Doctor of Philosophy in Computer Science to Indian Statistical Institute, Kolkata, is a record of bonafide research works carried out by him under our direct supervision and guidance.

We consider that the thesis has reached the standards and fulfilling the requirements of the rules and regulations relating to the nature of the degree. The contents embodied in the thesis have not been submitted as a whole or as a part for the award of any degree or diploma or any other academic award anywhere before.

24.08.2021

Assoc. Prof. Sushmita Ruj
Cryptology and Security Research Unit,
Indian Statistical Institute, Kolkata
203 Barrackpore Trunk Road,
Kolkata, West Bengal, India - 700108
&
Senior Research Scientist,
CSIRO Data61, Australia

Prof. Bimal Kumar Roy 25.8.21
Applied Statistics Unit
Indian Statistical Institute, Kolkata
203 Barrackpore Trunk Road
Kolkata, West Bengal, India - 700108

List of Publications/manuscripts

1. **Laltu Sardar**, Binanda Sengupta and Sushmita Ruj. Efficient Keyword Search on Encrypted Dynamic Cloud Data. *Advances in Mathematics of Communications*. Manuscript Submitted

2. **Laltu Sardar** and Sushmita Ruj. FSPVDsse: A Forward Secure Publicly Verifiable Dynamic SSE scheme. *Provable Security - 13th International Conference, ProvSec 2019*, pages 355-371, 2019. DOI: 10.1007/978-3-030-31919-9_23

3. **Laltu Sardar** and Sushmita Ruj. Verifiable and Forward private Conjunctive keyword Search from DIA Tree. *Designs, Codes and Cryptography*, Manuscript Submitted

4. **Laltu Sardar** and Sushmita Ruj. The Secure Link Prediction Problem. *Advances in Mathematics of Communications*, 13(4):733-757, 2019. DOI: 10.1007/978-3-030-31919-9_23

5. **Laltu Sardar**, Gaurav Bansal, Sushmita Ruj and Kouichi Sakurai. Securely Computing Clustering Coefficient for Outsourced Dynamic Encrypted Graph Data. *ComsNets 2021- 13th International Conference on COMmunication Systems & NETworkS*, pages 465–473, 2021. DOI: 10.1109/COMSNETS51098.2021.9352809

ACKNOWLEDGMENTS

This thesis is the beginning of a new journey towards my research career. I have learnt a lot during the period of obtaining my Ph.D. This thesis has been kept on track and been seen through to completion with the encouragement and support of numerous people including my well-wishers, my friends, teachers, and various institutions. Now, it is a pleasant task to express my thanks to those who made this thesis possible and an unforgettable experience for me.

My sincere and heartfelt gratitude and appreciation to my supportive supervisor **Assoc. Prof. Sushmita Ruj** for her enthusiastic encouragement, patient guidance, and useful critiques in my research work. Her valuable and constructive suggestions help me to efficiently develop my research works. Her willingness to give time so generously has been very much appreciated. She has always been a major source of support when things would get a bit discouraging. Without her persistent help, inspiration, and immense support, this thesis would not have been realized.

I would like to express my deep gratitude to my co-supervisor, **Prof. Bimal Kumar Roy** for his constant immense support, both directly and indirectly, throughout my Ph.D. years . His financial support on the one hand helped me to acquire such knowledge from different conferences and learning programs, on the other hand, his contribution to my thesis submission is undeniable.

I would like to give special thanks to **Prof. Avishek Adhikari**. He guided and encouraged me all the time to choose the path I am on now. It was a great experience to work with **Dr. Binanda Sengupta** who helped me to learn a lot.

I would like to acknowledge the assistance of **Prof. Kouichi Sakurai** who financially supported me to visit Kyushu University. I would like to extend my sincere thanks to Dr. Sabyasachi Dutta. His presence made my visit to the university easier and memorable. It was a good experience working with **Mr. Mamun Shaikh** and **Mr. Gaurav Bansal** during their internship at our institute. I would like to acknowledge the effort that I received from them.

I would like to express my sincere thanks to all faculty members of Computer and Communication Sciences Division and Applied Statistics Division, specially CSRU and ASU. I have gathered an ample amount of knowledge from them during my Ph.D. tenure.

I am thankful to **Mr. Avishek Majumder** for reviewing the thesis so carefully and helping me to make it better. I would like to thank my friends; **Mr. Subhadip Singha**, **Mr. Jyotirmoy Basak**, **Mr. Mostafizar Rahman**, **Mr. Prabal Banerjee**, **Mr. Aniruddha Biswas**, **Ms. Nayana Das**, **Mr. Soumya Das**, **Mr. Ram Govind**, **Mr. Nishant Dhanaji Nikam**, **Ms. Subhra Majumdar**, **Mr. Pritam Chattopadhyay**, **Mr. Diptendu Chatterjee**, **Mr. Samir Kundu**, and many more with whom I have shared moments of not only big excitement but also of deep anxiety. In a process that is often felt as tremendously solitaire, their presence was very important. I am also very grateful to

have seniors like **Dr. Srimanta Bhattacharya**, **Dr. Sanjay Bhattacherjee**, **Dr. Nilanjan Dutta**, **Dr. Subhabrata Samajder**, **Dr. Koushik Majumder**, **Mr. Amit Jana**, **Dr. Avijit Dutta**, **Dr. Aswin Jha**, **Dr. Sourav Sengupta**, and many more who helped me at different times in different ways. Discussing a variety of topics with all friends and seniors during the break, visiting different conferences and shorts trips are memorable.

Some special words of gratitude go to non-academic staffs of ASU, CSRU, accounts section and Dean's office, who I am contacted with. Their work helped hand me to process fast when it was necessary.

I would like to offer my heartfelt regards to Indian Statistical Institute for providing all the facilities and support required for my research throughout these years. During these years, I have attended conferences and visited research labs in India and abroad as well — that has been possible due to the support I obtained from the Dean of Studies (ISI Kolkata) and the NetApp Faculty fellowship Program, the Samsung Global Research Outreach (GRO) Program, etc.

Finally, my deep and sincere gratitude to my family, **my parents** and **my sisters**, for their continuous support, love, and help. I am very thankful to my wife **Sudeshna** who always tried to be with me during the journey of my entire Ph.D. They always encouraged me to not only explore new directions in life and but also seek my own destiny. Without them, this journey would not have been possible for me.

Date: August 24, 2021 Laltu Sardar

ABSTRACT

Nowadays, cloud computing and storage services have become crucial in everyone's life either directly or indirectly. However, data stored in untrusted cloud servers is prone to attacks by the server itself. In case, the client's data is sensitive or confidential, the outsourced data need to be stored in an encrypted form. But data encryption poses a challenge for querying over the encrypted outsourced data. *Queryable encryption* (QE) schemes on encrypted data allow clients to request a query over the encrypted data stored in the cloud. This cloud can be either honest-but-curious or malicious. There might be different types of data and different types of queries over them. One of the primary targets of a QE scheme is to keep the data and queries confidential. Moreover, queries should be efficient even for large data with update support.

Dynamic Searchable Encryption (DSE) is a queryable encryption that deals with dynamic text data. In a forward private DSE scheme, adding a keyword-document pair does not reveal any information about the previous search result with that keyword. Whereas, a backward private DSE scheme ensures that search queries do not reveal the information about the deleted file identifiers. To be protected from file-injection attacks, a DSE scheme is desired to be forward and backward private.

In this thesis, at first we propose a new and efficient DSE scheme Trids, based on an efficient data structure, for cloud data that achieves better security guarantees and improved efficiency compared to popular DSE schemes. Then, we propose a generic publicly verifiable DSE scheme Srica that makes any forward private DSSE scheme verifiable without losing forward privacy. Moreover, we design a forward private DSE scheme Blasu that supports conjunctive keyword search. At the heart of the construction is our proposed data structure called *DIA Tree* which is an authentication tree that efficiently returns both membership and non-membership proofs.

When the data is a graph, we study the secure link prediction that predicts which new interactions between members are most likely to occur in the near future. We use the number of common neighbors for prediction. We present three algorithms for the secure link prediction problem. Finally, we address the problem of computing clustering coefficient securely. The clustering coefficient is a measure of the degree, to which, the nodes in a graph cluster or associate with one another. We design a scheme Gopas to perform the query on an outsourced encrypted appendable graph data.

All the above-mentioned proposed schemes are provably secure. Moreover, we implement prototypes of most of the schemes and tested them with real-life data. The implementation results show that the schemes are practical even for a large database.

Contents

List of Figures

List of Tables

List of Definitions

List of Abbreviations

CSPRNG	Cryptographically Secure Psuedo-Random Number Generators
DCCE	Dynamic Clustering-Coefficient queryable Encryption
DIA Tree	Dynamic Interval Accumulation Tree
DSE	Dynamic Searchable Encryption
DSSE	Dynamic SSE
GC	Garbled Circuit
MGC	Maximum Garbled Circuit
PRF	Psuedo-random function
PRNG	Psuedo-Random Number Generators
PRP	Psuedo-random permutation
QE	Queryable Encryption
SE	Searchable Encryption
SLP	Secure Link Prediction
SPE	Searchable Public-key Encryption
SSE	Searchable Symmetric Encryption
TDL	Tri-Directional Linked-list
VDSSE	Verifiable Dynamic Searchable Symmetric Encryption
VSSE	Verifiable Searchable Symmetric Encryption

Chapter 1

Introduction

Nowadays, with the growing volume of data, cloud storage has become an important requirement where the cloud users (clients) can outsource their bulk data to the cloud servers (cloud). Cloud computing and storage services have become crucial in everyone's life either directly or indirectly. Google Cloud Platform, Microsoft Azure, Amazon Web Services (AWS), etc. offer outsourced computing to their clients having less computational power for various algorithms on stored data. People, who have less storage and are required to store a large amount of data and search over them, uses cloud storage services like IBM Cloud, Google Drive, Dropbox, Microsoft Onedrive, etc. As technology is spreading all over the world fast, the size of data is increasing very fast. If we consider only graph data, [4] shows a historical development of large-scale graph algorithms till 2016. It is quite difficult to store the data in local machines. So the data owner needs to outsource it to the cloud service providers.

Traditionally, the data is transferred from the data owner to the cloud service provider using end-to-end encryption schemes like symmetric key encryption. In such a scenario, "no information" (except the leakage allowed by the encryption scheme) is leaked to a listener of the channel. However, data is completely revealed to the cloud service provider.

Data stored in untrusted cloud servers are prone to attacks by the server itself. If the data is sensitive, confidential, and valuable like medical records, defense data, personal and biometric information, etc., it can be misused by the cloud service providers. CyberSecurity Watch Survey [23] shows that 53% of cyberattacks, reported in 2012, were a result of an insider attack. According to the report, 63% of the attacked data was due to unintentional exposure of private or sensitive data. Again, the cloud storage may be hacked by outside hackers as it happened. [23] shows 53% of electronic crime events were by outside-hackers while in 17% case attackers were unknown. These can be prevented only if the data owner uploads only encrypted data.

In order to protect confidential information, clients need to outsource data in encrypted form. This makes searching on data quite challenging. Data encryption poses a challenge for computation and searches over encrypted data. For example, trivially the data owner can upload all data in encrypted form to the cloud. Whenever some data-related query is needed, the data owner can download all data, does necessary computations, and re-uploads the re-encrypted data. But this is very inefficient and does not serve the purpose of cloud service. Thus, we need to keep the data stored in the cloud in encrypted form and as well as keep a mechanism to compute the required computation on the cloud.

1.1 Queryable Encryption (QE)

Queryable Encryption (QE) schemes on encrypted data allow a client to request a query over the encrypted data outsourced to the cloud. There might be different types of data and different types of queries over them. However, in a typical queryable encryption scheme, there are three entities in general– owner, cloud, and user (see Figure 1-1). We describe them as follows.

Owner is an entity that owns the data. It generates the required keys and encrypts the data with a suitable data structure. Later it uploads the encrypted data to the cloud server.

Cloud is the storage and computational service provider. It stores the encrypted data and performs a query over it on request from the user.

User is an entity that performs a query on the encrypted data. It requests a query to the cloud and gets the required result and decrypts it if necessary. The owner also can be a user.

Moreover, we assume that there is a secure communication channel between any two entities.

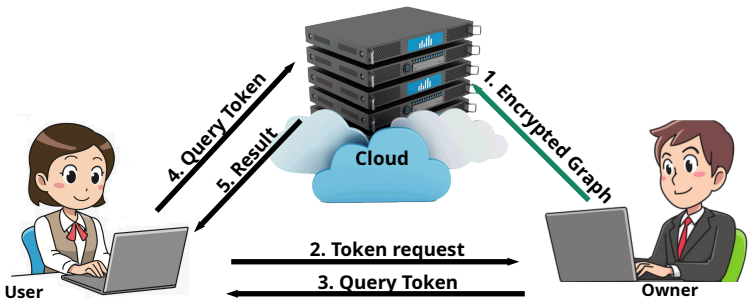


Figure 1-1: The system model of a typical queryable encryption scheme

1.1.1 Adversarial model

In the setting of queryable encryption schemes, the communication between the entities is assumed to be done via a secure channel. However, the entities are assumed to have different power.

The **Owner** of the database is assumed to be honest and trusted throughout this thesis.

The **User** is also assumed to be trusted. It only uses the services provided by the cloud and the owner.

The **Cloud** stores the encrypted data and it wants to know about the database and queries. The assumption of the cloud being honest is not preferred. This is because even if the cloud is honest,

the data can be hacked by some other third-party attacker. However, in the case of sensitive data, it is desirable not to consider the cloud to be honest. In this thesis, the cloud is considered either honest-but-curious or malicious. Moreover, we assume the cloud is the only adversary.

An honest-but-curious cloud stays honest and executes the protocol correctly. However, it wants to learn about the database from the leakage information it gets by executing the queries. A malicious cloud server also wants to learn about the database. But it may not follow the protocol and still remains undetected to its clients (owner and users). In both cases, the cloud is assumed to have probabilistic polynomial-time execution power.

1.1.2 Goal of a queryable encryption scheme

Confidentiality One of the primary targets of a queryable encryption scheme is to keep the data and queries confidential. When the data is being uploaded to the cloud server in encrypted form, the cloud server should not get any meaningful information about the data. Moreover, the cloud server should not get any meaningful information from the queries as well.

Scalability While a queryable encryption scheme has confidentiality, the cloud, directly or indirectly, cannot use the data to sell them in any form. To provide the service, the cloud server has to charge from owner/users. Since "There ain't no such thing as a free lunch", the owner will always want the capability to outsource a large amount of data. So, any queryable encryption should support outsourcing large data and querying over that.

Efficiency is another concern of a queryable encryption scheme. Keeping the data and the queries confidential is not enough. We consider the client to be computationally weak, but the cloud has a large amount of storage and high computational power. A queryable encryption scheme should be efficient for both clients and cloud servers for large databases.

Update support In most enterprises, data is dynamic. It changes very frequently over time. Supporting the updates over outsourced encrypted data is necessary and quite challenging without leaking much meaningful information about the queries and the data.

In the next sections of this chapter, we categorize queryable encryption schemes with different aspects.

1.1.3 QE with different update type

It is easier to design a queryable encryption scheme when the data is *static*. In most cases, the expected result may be pre-computed. However, nowadays, in many applications, data is not static and requires to be changed over time. So it is important to consider if data may be updated. Such data is called *dynamic data*.

The second important consideration is what type of updates are permissible. Appendable schemes allow only the append of new entries. An encryption scheme that can support the deletion of some existing data together with append is called a dynamic encryption scheme. Moreover, there are queryable encryption schemes that allow only the edit of some entries.

1.1.4 QE with different number of owners, users, and servers

A database might have either single or multiple owners. They are able to update the data if required. Secondly, it is important who can query. For an online database of an enterprise, there might be multiple managers of the database, we can say each of them is an owner. Similarly, there might be multiple employees who can request queries over it. In such a scenario, owners can also perform queries over the encrypted data. So, the presence of multiple owners and multiple users is very much practical. In some schemes, only the owners are allowed to request a query. Some of them are single-owner schemes. In some other types of applications, the owner or the user can be added or revoked.

The number of servers is also important. In some schemes, the owner uploads its data to multiple servers for load balancing, robustness, and privacy. The data is distributed on several servers. This keeps a major portion of the data hidden from a server. A server that mainly keeps the data, is called a *cloud server* (CS) or *cloud service provider*. In some cases, instead of distributing data, the computation is distributed. This type of server is called *proxy server* (PS) [97] which is a *computational service provider*. There may be a good amount of interaction between these cloud and proxy servers [83].

1.1.5 QE with different Encryption techniques

As symmetric key encryption (SKE) schemes are comparatively faster than public-key encryption (PKE) schemes with the same security, SKE schemes are preferable. Again, in queryable encryption, since we generally try to outsource large data, the application of SKE schemes is more

practical. However, there are many queries (for example, Order-Preserving Encryption [92]) on many types of data, where it is hard to provide a scheme using SKE schemes only. Sometimes, using PKE schemes makes the queryable scheme simpler with greater security.

For example, if the data is a set of numbers and the query is finding the shortest one, an order revealing encryption scheme is preferable. When a verifiable conjunctive search is considered over dynamic data, it is difficult to design such a scheme with SKE. Again, for single keyword search schemes in text data, SKE schemes are used in general. As we see that, in a queryable encryption scheme for text data, all documents in the source data are always encrypted with some SKE schemes. However, the encrypted index for query may be generated with either SKE ([32, 48, 46]) or PKE ([11, 61]) or mix of both ([44, 6, 45, 97]). For other types of data also, the use of PKE schemes is quite popular.

1.1.6 QE with different data and query type

The design of a queryable encryption scheme highly depends on data and query type. There might be different types of data and different types of queries over them. For example, when data is a set of documents, where each document consists of a set of keywords, we can query for the set of documents containing a single keyword, or a set of keywords, or some combination of them. In case, when the data is a set of numbers, we can search for a certain range. When data is a graph, one can ask to execute graph algorithms, and so on.

QE on Text data The type of data is very crucial for a queryable encryption scheme. Depending on the type of data, specific data structures are used. The type of query also depends on the data. Most of the works on queryable encryption are on text data. The text data consists of a set of documents, where each document consists of a set of keywords from a dictionary. In most of the paper, this dictionary is considered to be predefined and fixed, whereas in a few, it is considered flexible to be any string [74].

Searchable encryption (SE) schemes on encrypted data allow a client (data owner) to search for documents (or files) matching given keywords over a collection of encrypted documents stored in the cloud. In a typical SE scheme, a client encrypts a set of documents and uploads them to the cloud server. Later, the client sends the (encrypted) search query for a keyword to the cloud server, and the server replies with the query results (i.e., the encrypted documents containing that particular keyword). Finally, the client decrypts the encrypted documents. Thus searchable encryption is queryable encryption that deals with text data.

In general, when data is uploaded to the cloud server in unencrypted form (like dropbox, google drive, etc.), they are sent via a secure communication channel. The server can read all data, it can make search indices corresponding to them. So, in that case, the server can respond to a query very fast. However, it is not reasonable to outsource plaintext to the malicious cloud as it learns the complete data. But, when a client sends encrypted data, the cloud cannot read it, cannot make a search index and looses the ability to search. So, in a searchable encryption scheme, the data owner has to make the search index. These search indices are in encrypted form and send together with the encrypted set of documents. For text data, the documents are encrypted with some symmetric encryption scheme, and then an encrypted search index is generated with the identifiers of the documents. So, most of the searchable encryption schemes deal with encrypted search index generation.

One of the primary objectives of designing an SE scheme for cloud data is to reveal as little information to the cloud server as possible. A naive solution is the following. For every search query, the client downloads the complete encrypted data, decrypts them, and performs the search on the data. If some update is needed, then do the same on the downloaded data and uploads it back. However, downloading the whole data for each search query is impractical and defeats the purpose of outsourcing data to the cloud server. SE schemes on encrypted data handle keyword searches more efficiently than this naive solution.

When an SE scheme is designed with symmetric key encryption it is called *Searchable Symmetric Encryption* (SSE), whereas it is designed with public key encryption it is called *Searchable Public-key Encryption* (SPE). However, since public-key encryption requires greater computational power than symmetric key encryption, the use of public-key encryption is generally avoided while designing a searchable encryption scheme.

QE on Dynamic text data We can see that uploading data to the cloud in an encrypted form is not enough. In the case of dynamic data, the client may need to add or delete data. In such a scenario, the SSE scheme should be able to support database updates. An SE (or SSE) scheme, that supports such updates, is called a *Dynamic SE* (Dynamic SSE) scheme.

A DSE scheme can have the following leakages.

1. *Access Pattern Leakage:* This reveals the set of document identifiers accessed by a query. For example, the cloud server can know the set of document identifiers containing the searched keyword from a search query.

2. *Search Pattern Leakage:* This reveals the relation between two keywords being searched.

For example, the search pattern indicates whether a search query is repeated or not.

3. *Size Pattern Leakage:* This tells the size of the encrypted documents stored in the database.

For any DSSE scheme, it is desirable to reduce these leakages. However, most of the schemes have size pattern leakage as it leaks very little information.

There are plenty of works on SSE ([32], [86], [26], [93], [27]) as well as DSSE ([87], [42], [21], [35], [47], [100], [79], [60], [15], etc.).

DSE schemes are designed depending upon the type of data and query. When the data is a set of documents, each containing a set of keywords, some popular queries over them include single keyword search, conjunctive or Boolean search on a set of keywords, etc. In a *single keyword search* SE (or SE) scheme, given a keyword, the cloud returns the set of documents that contains it. In a *conjunctive keyword search* SE (or conjunctive SE) scheme, given a set of keywords, the cloud returns the set of documents that contains all of them.

Query over Encrypted Graph Social networks have become an integral part of our lives. These networks can be represented as graphs with nodes being entities (members) of the network and edges representing the association between entities (members). As the size of these graphs increases, it becomes quite difficult for small enterprises and business units to store the graphs in-house. So, there is a desire to store such information in cloud servers.

It is a good challenge to execute graph algorithms on an encrypted graph without revealing the graph structure and other information. This is because, in most of the graph algorithm, it is required to visit random nodes. Sometimes, it is required to visit all nodes in the graph, for example, in Kruskal's algorithm to find a minimum spanning tree. If each node of the graph is encrypted separately, information to decrypt any node should be given.

The process can be interactive as well. After each step client and cloud can communicate and the client gives only the necessary information for the next round. However, this might require a huge communication cost. From a security point of view, it is better than revealing all of the graph. However, the second approach does not work for all types of graphs.

In order to protect the privacy of individuals (as is now mandatory in EU and other places), data is often anonymized before storing in remote cloud servers. However, as pointed out by Backstrom *et al.* [7], anonymization does not imply privacy. By carefully studying the associations between members, a lot of information can be gleaned. Thus, encryption of graph is necessary before outsourcing to a third-party cloud server.

Since, in this active world, the relationships among individuals are changing continuously, the graphical structures need to be updated. In such a scenario, the addition of new edges or nodes, and the deletion of old ones are very frequent. This increases the complexity of social networks and makes it highly dynamic. Thus, the design of a graph encryption scheme should support dynamic updates efficiently.

A client can request different queries on the encrypted graph. Its target is to get as much information as possible, leaking as little information as possible to the cloud server. There are some basic queries that we think fundamental queries of a graph. Irrespective of the type of graph a client can query them over the outsourced database. For example, some basic queries include the following.

- *Neighbor query*: In this query, given a vertex, the cloud returns the set of vertices adjacent to it.
- *Vertex query*: Given a vertex, the cloud returns whether the vertex is present or not.
- *Edge query*: In this query, given two vertices, the cloud returns whether the edge between them is present or not. The query is also called *adjacency query*.
- *Degree query*: In this query, given a vertex, the cloud returns the number of vertices adjacent to it.

There are many other graph algorithms that are queried. For example, *Shortest Distance queries* where, given two vertices, the cloud returns the shortest distance between them. In some other variants, the cloud returns the shortest path together with the shortest distance. Shortest distance query, has been studied in different ways in [84], [65], [97], etc.

Chase and Kamara [27] studied adjacency queries on graphs and focused subgraph queries on labeled graphs which returns all the vertices that are connected either to or from the given vertex.

Lai and Chow [34] presented a bipartite graph encryption scheme. Since, assuming two distinct sets– the set of keywords and the set of documents, they mapped searchable encryption to a bipartite graph encryption problem and provided a solution to a single keyword search scheme.

Other types of data In applications, such as social networks, the data may be of mixed type. Each node may contain some set of information that is kept as a document set. Chase and Kamara [27] studied query on data.

Sometimes, the data may be an ordered set of elements, for example, a set of integers, a set of words. In such a scenario, range query [113] is quite popular where given two elements, all members of the set between them are returned.

1.1.7 QE with different type of adversaries

Semi-honest Server and Forward Privacy Most of them consider the cloud server to be semi-honest, in other words, honest-but-curious. An honest-but-curious cloud server follows the protocol correctly, but it wants to extract information about the plaintext data and the queries. In presence of a semi-honest cloud server, in a DSE scheme, updating the database may reveal the relation between the updated set of keywords and the previous search result. In such a scenario, the file injection attack ([110]) can be performed by a curious cloud server. In this attack, the client encrypts and stores files sent by the server. From these added files, the server recovers keywords from future queries. However, forward privacy can protect a DSE scheme from file injection attacks. A forward private DSE scheme does not leak any information about the previous search results when new documents are added. This attack has forced researchers to think about DSE schemes to be forward private.

Backward Privacy The backward privacy of a searchable encryption scheme ensures that search queries do not reveal information about the deleted identifiers. Bost et al. [18] defines three variants of backward privacy with different measures of security. The categorization is only for text data which consists of a set of documents each containing a set of keywords. Thus the categorization is valid only for dynamic searchable encryption schemes. Given a keyword w and a time interval between two search queries on w, we give their definitions according to [113] as follows.

Type-I *backward privacy*. It leaks information about when new files containing w were inserted and the total number of updates on w.

Type-II *backward privacy*. It leaks information of Type-I. It additionally leaks when all updates (including deletion) related to w occurred.

Type-III *backward privacy*. Apart from the leakages of Type-II, it also leaks exactly when a previous addition has been canceled by which deletion.

In addition, [113] gives another kind of backward privacy Type-I$^-$ backward privacy which is stronger than Type-I backward privacy.

Type-I$^-$ *backward privacy*. Given the same information as above, it leaks the files that currently

match w and the total number of updates for w.

Malicious Cloud Server and Verifiable Query However, if a curious cloud server becomes malicious, it does not follow the protocol correctly. In the context of search, it can return only a subset of results, instead of all the records of the search. It can also return incorrect information if the client does not check. Explicitly, it might think a document contains a certain keyword. It may not return the actual search result for monetary or other benefits. For example, if some large sets of documents are not accessed by the client for a long period of time, then the cloud server can delete them from the database which can reduce the cost of storage as well as the cost of computation. So, there is a need to verify the results returned by the cloud to the user. The results should be derived from the actual state of the database and complete.

A verifiable searchable encryption scheme guarantees correctness and completeness of the search result even when the cloud server is malicious. A cloud server not only sends the search result but also proof that the result is correct. An SSE scheme for static data where the query results are verifiable is called Verifiable SSE (VSSE). Similarly, if the data is dynamic the scheme is said to be a verifiable dynamic SSE (VDSSE).

There are single keyword search VSSE schemes that are either new constructions supporting verifiability or design techniques to achieve verifiability on the existing SSE schemes by proposing generic algorithms. VSSE with single keyword search has been studied in [24], [28], [63], [62], etc. In [91], [95], [96] etc., VSSE scheme with conjunctive query has been studied. Moreover, there are also works that give VDSSE scheme for both single keyword search ([66]) as well as complex query search including fuzzy keyword search ([112]) and Boolean query ([44]). Most of them are *privately verifiable*. A VSSE or VDSSE scheme is said to be *privately verifiable* if the only user, who receives the search result, can verify it. On the other hand, a VSSE or VDSSE scheme is said to be *publicly verifiable* if any third party, including the database owner, can verify the search result without knowing its content.

There is also literature on public verifiability. Soleimanian and Khazaei [85] and Zhang et al. [109] have presented SSE schemes that are publicly verifiable. VSSE with Boolean range queries has been studied by Xu et al. [102]. Though their verification method is public, since the verification is based on blockchain databases, it has an extra monetary cost. Besides, Monir Azraoui [6] presented a conjunctive search scheme that is publicly verifiable. In the case of dynamic database, the publicly verifiable scheme by Jiang et al. [44] supports Boolean Query and that by Miao et al. [66] supports single keyword search.

1.2 Organization of Thesis and Our Contribution

The rest of the chapters of the thesis are organized as follows.

We discuss the required preliminary topics in Chapter 2. In this chapter, we give a short description of the notations and cryptographic tools used in the next chapters. Other notations and tools specific to a chapter are mentioned in the respective chapter.

Then, In Chapter 3, we provide a detailed survey of the existing literature on queryable encryption schemes.

We summarize our contribution in other chapters of this thesis as follows.

1. **A Dynamic Symmetric Searchable Encryption Scheme** We see that, searchable encryption schemes enable a client to search and retrieve the cloud data (based on the keywords present in the data) when the data is encrypted. Whereas dynamic searchable encryption schemes allow the client to search over the encrypted cloud data even when new documents are added to or deleted from the encrypted data. There is a trade-off between security (that is measured in terms of information leaked to the cloud) and the efficiency of dynamic searchable encryption schemes. Stronger security guarantees often come at a cost of less efficiency.

   In Chapter 4, we propose a new and efficient dynamic searchable encryption scheme Trids for cloud data that achieves better security guarantees and improved efficiency compared to popular dynamic searchable encryption schemes. Our scheme Trids uses an efficient data structure that reduces storage, lookup (search) time, and database modification time. We build a prototype of our scheme and experiment on large real-life datasets. We show that our scheme Trids performs better than the existing schemes which provide similar (or weaker) security.

2. **A Forward Secure Verifiable DSSE Scheme** When the data supports update, in a searchable encryption scheme, the information leakage increases which makes the scheme vulnerable to the cloud. A scheme can be protected from this vulnerability, only if it must be forward private in which adding a keyword-document pair does not reveal any information about the previous search result with that keyword. Again in the SSE setting, when the cloud server is malicious, meaning that it can alter the data, it becomes difficult to achieve forward privacy. Verifiable dynamic SSE requires the cloud server to give proof of the result of the search query. The data owner can verify this proof efficiently.

In Chapter 5, we propose a generic publicly verifiable dynamic SSE scheme Srica that makes any forward private DSSE scheme verifiable without losing forward privacy. The proposed scheme Srica does not require any extra storage at the owner-side and requires minimal computational cost as well for the owner. Moreover, in this chapter, we compare our scheme Srica with the existing results and show that our scheme is practical. The work presented in this chapter has been accepted in a conference proceeding [82].

3. **A Forward Secure Verifiable Conjunctive Search Scheme** There has been a fair deal of work on designing forward private DSE schemes in presence of honest-but-curious cloud servers. However, a malicious cloud server might not run the protocol correctly and still want to be undetected. In a verifiable DSE, the cloud server not only returns the result of a search query, but also provides proof that the result was computed correctly. The client accepts a result from a malicious cloud only when this proof is efficiently verified.

   In Chapter 6, we design a forward private DSE scheme Blasu that supports conjunctive keyword search. At the heart of the construction is our proposed data structure called *DIA Tree*. DIA Tree is an accumulator-based authentication tree that efficiently returns both membership and non-membership proofs. Using the DIA tree, we can convert any single keyword forward private DSE scheme to a verifiable forward private DSE scheme that can support conjunctive query as well. Our proposed scheme Blasu has the same storage as the base DSE scheme and minimal computational overhead at the client-side. We show the efficiency of our design by comparing it with existing conjunctive DSE schemes. The comparison also shows that our scheme is suitable for practical use.

4. **The Secure Link Prediction** In searchable encryption, generally the data is a set of collection of documents and the queries are some form of keywords. When the data is a graph, the queries also becomes different. Since, in this century, we belong to one or more communities and connected with peoples around. To represent these connections, we use graph structures that help to analyze the community. When small enterprises, with low storage and computational power, want to outsource their data and computation to a third-party cloud, only the anonymization does not help to protect individual data privacy. Moreover, fear of getting the data leak and misuse by unauthorized persons forces the data to be encrypted before outsourcing which makes the cloud difficult to perform queries on it. It is necessary to bring a technique that allows queries to be performed on encrypted outsourced data without leaking meaningful information.

   The link prediction problem, which predicts which new interactions between members are most likely to occur in the near future, is a well-studied problem for graph data. In Chapter 7,

we study the secure link prediction in encrypted outsourced graph, using the number of common neighbors for prediction. We present three algorithms for the secure link prediction problem. In this chapter, we design prototypes of the schemes and formally prove their security. Moreover, we execute our algorithms in real-life datasets. The work presented in this chapter has been accepted as a journal article [83].

5. **The Secure Clustering Coefficient Query in Outsourced Encrypted Dynamic Data** When the data is social network data, the clustering coefficient is such a property that quantifies the abundance of connected triangles in a network. In Chapter 8, we introduce the clustering coefficient query and design a scheme Gopas to perform the query on an outsourced encrypted appendable graph data. We show that the designed scheme Gopas is secure under the chosen-query attack. Moreover, we implement a prototype of the scheme and tested it with real-life data. The implementation results show that the scheme is practical even for a large database. The work presented in this chapter has been accepted in a conference proceeding [81].

1.3 List of Publications and Manuscripts

1. Laltu Sardar, Binanda Sengupta and Sushmita Ruj. Efficient Keyword Search on Encrypted Dynamic Cloud Data. *Submitted to a journal*

2. Laltu Sardar and Sushmita Ruj. FSPVDsse: A Forward Secure Publicly Verifiable Dynamic SSE scheme. *Provable Security - 13th International Conference, ProvSec 2019*, pages 355-371, 2019.

3. Laltu Sardar and Sushmita Ruj. Verifiable and Forward private Conjunctive keyword Search from DIA Tree. *Manuscript Submitted*,

4. Laltu Sardar and Sushmita Ruj. The Secure Link Prediction Problem. *Advances in Mathematics of Communications*, 13(4):733-757, 2019.

5. Laltu Sardar, Gaurav Bansal, Sushmita Ruj and Kouichi Sakurai. Securely Computing Clustering Coefficient for Outsourced Dynamic Encrypted Graph Data. *ComsNets 2021-13th International Conference on COMmunication Systems & NETworkS*, pages 465–473, 2021.

Chapter 2

Preliminaries

In this chapter, we briefly discuss some preliminaries and cryptographic tools used in this thesis.

2.1 Notations

The set of n-bit strings is denoted by $\{0,1\}^n$, whereas $\{0,1\}^*$ is the set of all finite length bit strings. $|S|$ denotes the cardinality of the finite set S. For a positive integer n, $[n]$ denotes the set $\{1,2,\ldots,n\}$. By $out \leftarrow \mathcal{A}(in)$, we mean a probabilistic algorithm $\mathcal{A}$, on input in, outputs out. $x \xleftarrow{\$} X$ denotes that x is chosen uniformly at random from the set X. Some notations used in the thesis are given in the Table 2.1.

Table 2.1: Notations

Symbols	Meaning
$\mathcal{W}$	A set of keywords/ dictionary
λ	The security parameter
$\mu(\lambda)$	A negligible function over λ
$[n]$	The set of integers $\{1,2,\ldots,n\}$
Σ	A dynamic searchable encryption scheme
Σ_s	A result revealing static SSE scheme
Σ_f	A forward private dynamic searchable encryption scheme
Ψ_s	A generic verifiable static SSE scheme
Ψ_f	A verifiable forward private dynamic SSE scheme

2.2 Stateful algorithm

Definition 2.1 (Stateful algorithm)**.** *An algorithm is said to be a* stateful algorithm *if it stores its previous states and use them to compute the current state.*

2.3 Negligible function

A negligible function is used to show the security of a scheme. A scheme is secure if the success probability of some formally specified type attack, carrying out by every probabilistic polynomial-time (PPT) adversary, is negligible.

Definition 2.2 (Negligible function)**.** *A function* $\mu : \mathbb{N} \leftarrow \mathbb{R}$ *is said to be* negligible *[50], if* $\forall c \in \mathbb{N},\ \exists N_c \in \mathbb{N}$ *such that* $\forall n > N_c,\ \mu(n) < n^{-c}$.

2.4 Cryptographically secure pseudo-random number generator (CSPRNG)

Cryptographic algorithms require long random numbers which can be regenerated. Random numbers, from the source of like time, movement of cursor, etc., are difficult to regenerate. However, a random number that can be regenerated, is no longer complete random. So, with little compromise of uniform randomeness, psuedo-random number generators (PRNG) generate longer, "uniform-looking" (or "pseudo-random") output string from a shorter uniform string seed. If outputs of a PRNG stay indistinguishable from true random numbers, except with negligible probability, it is said to cryptographically secure. Formally, we define it as follow.

Definition 2.3 (CSPRNG)**.** *A deterministic polynomial-time algorithm* $G : \{0,1\}^{\lambda} \to \{0,1\}^{p(\lambda)}$*, for some polynomial* p*, is said to be a* cryptographically secure pseudo-random number generator *(*CSPRNG*) [50], if*

1. *It stretches the length of its input i.e.,* $p(\lambda) > \lambda$ *for any* λ*,*
2. *Its output is computationally indistinguishable from true randomness, i.e. for any PPT algorithm A, here is a negligible function* μ *such that*

$$\left| \Pr_{s \leftarrow \{0,1\}^{\lambda}}[A(G(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{p(\lambda)}}[A(r) = 1] \right| < \mu(\lambda)$$

We call l the *expansion factor* of G and an input s the *seed* of it. Throughout, we denote a cryptographically secure pseudo-random number generator simply as a PRG (pseudo-random generator).

2.5 Psuedo-random function (PRF)

Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be a two input function, where the first input is the key denoted by k. When the key is fixed, it becomes a single input function $F_k : \{0,1\}^* \to \{0,1\}^*$ defined by $F_k(x) = F(k,x)$. If the length of the key is fixed to some integer n then input and output has length $l(n)$. Let $\texttt{Func}n$ be the set of all functions mapping $l(n)$-bit strings to $l(n)$-bit strings.

Thus, a keyed function F is said to be a pseudorandom function if, for any key k of length n, F_k is indistinguishable from any function f chosen at random from $\texttt{Func}_n$. We define formally as follows.

Definition 2.4 (Psuedo-random function)**.** *A collection of functions* $\{F_k : \{0,1\}^{l(|k|)} \to \{0,1\}^{l(|k|)}\}_{k \in 0,1^*}$, *where* $l : \mathbb{N} \to \mathbb{N}$, *is said to be psuedo-random ([39, 50]) if the following two conditions holds.*

1. Efficiency: *Given any* k *and* x *such that* $|x| = l(|k|)$, $F_k(x)$ *can be computed in polynomial-time .*

2. psuedo-randomness: *For any polynomial-time distinguisher* $\mathcal{D}$, $\exists$ *a negligible function* μ *such that:*

$$\left| \Pr_{k \xleftarrow{\$} \{0,1\}^n} [\mathcal{D}^{F_k(.)}(1^n) = 1] - \Pr_{f \xleftarrow{\$} \texttt{Func}_n} [\mathcal{D}^{f(.)}(1^n) = 1] \right| < \mu(\lambda)$$

2.6 Psuedo-random permutation (PRP)

The definition of psuedo-random permutation is similar to that of psuedo-random function (Def. 2.4). The only differences are that the function F_k is bijective, and is indistinguishable from permutation f chosen uniformly at random from $\texttt{Func}_n$, where $\texttt{Func}_n$ is the set of all permutations from $\{0,1\}^{l(n)}$ to $\{0,1\}^{l(n)}$. The definition can be given formally as follows.

Definition 2.5 (Psuedo-random permutation)**.** *A collection of permutations* $\{F_k : \{0,1\}^{l(|k|)} \to \{0,1\}^{l(|k|)}\}_{k \in 0,1^*}$, *where* $l : \mathbb{N} \to \mathbb{N}$, *is said to be psuedo-random ([39, 50]) if the following two conditions holds.*

1. Efficiency: *Given any* k *and* x *such that* $|x| = l(|k|)$, $F_k(x)$ *can be computed in polynomial-time .*

2. psuedo-randomness: *For any polynomial-time distinguisher $\mathcal{D}$, $\exists$ a negligible function μ such that:*

$$\left| \Pr_{k \xleftarrow{\$} \{0,1\}^n} [\mathcal{D}^{F_k(.)}(1^n) = 1] - \Pr_{f \xleftarrow{\$} \mathtt{Perm}_n} [\mathcal{D}^{f(.)}(1^n) = 1] \right| < \mu(\lambda)$$

2.7 Hash function

A Hash function is a computationally efficient function that maps arbitrary length bit-strings to some fixed length bit-strings, called hash-values.

Definition 2.6 (Hash function)**.** *A pair of probabilistic polynomial time algorithms (Gen, H) is said to be a hash function (with output length l) [50] if it satisfies the following.*

1. *Gen probabilistically outputs a key k, after taking a security parameter λ as input.*

2. *H takes as input a key k and a string $x \in \{0,1\}^*$ and outputs a string $H_s(x) = \{0,1\}^{l(\lambda)}$*

In cryptographic application, hash functions are generally used having a fixed key and a fixed output length. This makes the function looks like $H : \{0,1\}^* \to \{0,1\}^\lambda$. So, unless stated we assume has functions to be unkeyed (or fixed keyed).

A hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$ can have the following properties.

1. *Collision resistance:* A hash function H is collision resistant if, for any PPT adversary, it is computationally infeasible to find x, x' such that $x \neq x'$ and $H(x) = H(x')$.

2. *Second-preimage resistance:* A hash function H is second preimage resistant if, given a uniform x, it is computationally infeasible, for any PPT adversary, to find $x' \neq x$ such that $H(x') = H(x)$

3. *Preimage resistance:* A hash function H is preimage resistant if, given a uniform y, it is computationally infeasible for any PPT adversary to find a value x such that $H(x) = y$.

The *Cryptographic hash functions* considered throughout are unkeyed (or fixed keyed) and collision resistant. SHA-1, SHA-256, etc. are some example of cryptographic hash functions.

2.8 Message Authentication Codes (MACs)

When two parties want to communicate secretly, message authentication codes (MACs) generate tags that provides both message integrity as well as authenticity.

Definition 2.7 (Message Authentication Codes)**.** *A* message authentication code *(or* MAC*) consists of three PPT algorithms* $(\texttt{Gen}, \texttt{Mac}, \texttt{Vrfy})$ *as follows.*

1. $k \leftarrow \texttt{Gen}(1^\lambda)$*: On input the security parameter* 1^λ*, this key-generation algorithm outputs a key* k *with* $|k| \geq \lambda$*.*
2. $t \leftarrow \texttt{Mac}_k(m)$*: On input a key* k *and a message* $m \in \{0,1\}^*$*, this tag-generation algorithm outputs a tag* t*.*
3. $b \leftarrow \texttt{Vrfy}_k(m, t)$*: On input a key* k*, a message* m*, and a tag* t*, the deterministic verification algorithm* Vrfy *outputs a verification bit* b*, where* $b = 1$ *means valid and* $b = 0$ *means invalid.*

A MAC is said to be correct *if, for every* λ*, every key* $k \leftarrow \texttt{Gen}(1^\lambda)$*, and every* $m \in \{0,1\}^*$*, the condition* $\texttt{Vrfy}_k(m, \texttt{Mac}_k(m)) = 1$ *holds.*

The security of a MAC lies in the fact that from a set of message-tag pairs $\{(m_1, t_1), (m_2, t_2), \ldots, (m_n, t_n)$ it is computationally infeasible to find another pair (m', t') such that $m' \notin \{m_1, m_2, \ldots, m_n\}$ and $\texttt{Vrfy}_k(m', t') = 1$.

There are different ways to construct MACs. XOR MAC, CMAC, etc. are examples of pseudo-random function based MAC whereas HMAC is based on cryptographic hash functions.

2.9 HMAC

HMACs are message authentication codes based on keyed-hash. It is used to verify both message integrity and authenticity. SHA-256 or SHA-3, SHA-1 some example of HMACs.

2.10 Dynamic hash table

Hash tables are dictionary or table like data structures, based on hash functions, that stores and retrieve certain data efficiently. The data is stored as key-value pair, where key indicates the position

in the table. A hash table mainly supports look-ups of elements belong to a set.

Definition 2.8 (Dynamic hash table)**.** *[76] A hash table is a tuple of three algorithms* $\mathtt{HT} = (Insert, Search, Delete)$ *as follows.*

- $\mathtt{HT}.Insert(\mathtt{T}, (key, val))$*: It takes inputs a key-value pair* (key, val) *and store it as* $\mathtt{T}[key] \leftarrow val$*. We say as inserting key-value pair* (key, val) *to the hash table* T*. For simplicity we say* T *is the hash table.*
- $\mathtt{HT}.Search(\mathtt{T}, key)$*: Given* key*, it look-up in* T *and returns* $val \leftarrow \mathtt{T}[key]$*. If no such value exists then it returns* $val = null$*. We say it as searching* key *in the hash table* T*.*
- $\mathtt{HT}.Delete(\mathtt{T}, key)$*: Given a key* key*, it sets* $\mathtt{T}[key] \leftarrow null$ *if* $\mathtt{T}[key]$ *exists.*

There are different of implementations of hash tables. If we consider [30], then for a set of N elements, it uses $O(N)$ space and $O(1)$ expected amortized cost for elements insertions or deletions and have $O(1)$ expected query time for (non-)membership queries.

2.11 Multiset hash

A multiset is a finite unordered group of elements where an element can occur as a member more than once. A multiset hash outputs a single hash value for an unordered set of elements. Instead of giving multiple hash value for a set of strings, giving single hash value makes the communication shorter and storage requirement lesser.

Definition 2.9 (Multiset Hash)**.** *[29]. Let by* $M \sqsubset B$ *we mean a multiset* M *of elements of a countable set* B*. Let multiset union of two multisets* $M = \{m_1, m_2, \ldots, m_{|M|}\}$ *and* $M' = \{m'_1, m'_2, \ldots, m'_{|M'|}, \}$ *be defined as*

$$M \sqcup M' = \{m_1, m_2, \ldots, m_{|M|}, m'_1, m'_2, \ldots, m'_{|M'|}\}.$$

A triplet $(\mathsf{H}, +_{\mathsf{H}}, \equiv_{\mathsf{H}})$ *of PPT algorithms is said to be a* multiset hash *on* B *with security parameter* λ *when it satisfies the following properties:*

1. $\mathsf{H}(M) \in \{0,1\}^{\lambda}$, $\forall M \sqsubset B$ *(compression)*
2. $\mathsf{H}(M) \equiv_{\mathsf{H}} \mathsf{H}(M)$, $\forall M \sqsubset B$ *(comparability)*

3. $\mathsf{H}(M \sqcup M') \equiv_{\mathsf{H}} \mathsf{H}(M) +_{\mathsf{H}} \mathsf{H}(M')$, $\forall M, M' \sqsubset B$ *(incrementality)*

Clarke et al. [29] presented an incremental multiset hash function which is set-collision resistant.

2.12 Gap Diffie-Hellman (GDH) group

Definition 2.10 (GDH group). *[13]. Let* G *be a multiplicative cyclic group with prime order* p*. For* $a, b, c, \in \mathsf{Z_p}$*, given* $g, g^a, g^b, g^c \in \mathsf{G}$*, deciding whether* $c = ab$ *is called* Decisional Diffie-Hellman (DDH) *problem in* G*. Again, For* $a, b, \in \mathsf{Z_p}$*, given* $g, g^a, g^b \in \mathsf{G}$*, computing* $g^{ab} \in \mathsf{G}$ *is called* Computational Diffie-Hellman (CDH) *problem in* G*. The group* G *is said to be a* Gap Diffie-Hellman (GDH) group *if, the CDH problem is hard, but the DDH problem is easy in* G*.*

Definition 2.11 ((τ, t, ϵ)-GDH group). *[13]. The group* G *is said to be* (τ, t, ϵ)*-GDH group if, the DDH problem on* G *can be solved in at most time* τ *and no algorithm which runs in time at most* t *can break CDH on* G *with probability* $\geq \epsilon$*.*

2.13 Bilinear Map

Let G_1, G_2 and G_T be three (multiplicative) cyclic groups of prime order p. Let g_1 be a generator of G_1 and g_2 be a generator of G_2 i.e., $\mathsf{G}_1 =< g_1 >$ and $\mathsf{G}_2 =< g_2 >$. A map $\hat{e} : \mathsf{G}_1 \times \mathsf{G}_2 \rightarrow \mathsf{G}_T$ is said to be an *admissible non-degenerate bilinear map* if–

1. $\exists$ bilinearity i.e., $\hat{e}(u^a, v^b) = \hat{e}(u, v)^{ab}$, $\forall u \in \mathsf{G}_1$, $\forall v \in \mathsf{G}_2$ & $\forall a, b \in \mathsf{Z}_p$
2. $\exists$ non-degeneracy i.e., $\hat{e}(g_1, g_2) \neq 1$, and
3. $\exists$ efficiency i.e., $\hat{e}$ can be computed efficiently.

In our case, we consider $\mathsf{G}_1 = \mathsf{G}_2 = \mathsf{G}$, and $\mathsf{G} =< g >$. For our scheme we require the group G to be a GDH group. Let us consider the following bilinear map generating algorithms .

$(p, \mathsf{G}, \mathsf{G_T}, \mathsf{g}, \hat{\mathsf{e}}) \leftarrow \mathtt{BMGen}(1^\lambda)$: It is a PPT algorithm (bilinear map generator) that takes a security parameter λ as input and outputs a uniquely random tuples $(p, \mathsf{G}, \mathsf{G_T}, \mathsf{g}, \hat{\mathsf{e}})$ of bilinear pairing parameters.

$(p, \mathsf{G}, \mathsf{G_T}, \mathsf{g}, \hat{\mathsf{e}}) \leftarrow \mathtt{BMGGen}(1^\lambda)$: It is a PPT algorithm (bilinear map generator) that takes a security parameter λ as input and outputs a uniquely random tuples $(p, \mathsf{G}, \mathsf{G_T}, \mathsf{g}, \hat{\mathsf{e}})$ of bilinear pairing parameters where G *is a GDH group*.

2.14 Bilinear Pairings and Computational Problems

2.14.1 Discrete Logarithm Assumption

Definition 2.12 (Discrete Logarithm Assumption)**.** *Let λ be a security parameter and $(p, \mathsf{G}, \mathsf{G}_T, \hat{e}, g)$ be a uniformly randomly generated tuple of bilinear pairing parameters from $\mathtt{BMGen}(1^\lambda)$. It is said to be hold* discrete logarithm assumption *if, given g, g^a (where $a \xleftarrow{\$} \mathsf{Z}_p^*$), any probabilistic polynomial time adversary $\mathcal{A}$ can find a only with negligible probability, namely*

$$Adv_{\mathcal{A}}^{DLog}(\lambda) = \Pr_{a \xleftarrow{\$} \mathsf{Z}_p^*}[\mathcal{A}(g, g^a) = a] \leq \mu(\lambda)$$

i.e., no PPT algorithm $\mathcal{A}$ can output a with more than a negligible advantage.

2.14.2 Decisional Bilinear Diffie-Hellman (DBDH) Assumption

Definition 2.13 (Decisional Diffie-Hellman Assumption)**.** *Let λ be a security parameter and $(p, \mathsf{G}, \mathsf{G}_T, \hat{e}, g)$ be a uniformly randomly generated tuple of bilinear pairing parameters from $\mathtt{BMGen}(1^\lambda)$. We say that* decisional Diffie-Hellman assumption *holds in $\mathtt{BMGen}(1^\lambda)$ if, for any security parameter λ, and any probabilistic polynomial time (PPT) adversary $\mathcal{A}$*

$$Adv_{\mathcal{A}}^{DBDH}(\lambda) = \Pr[\mathcal{A}(g^a, g^b, g^c, \hat{e}(g,g)^{abc}) = 1] - \Pr[\mathcal{A}(g^a, g^b, g^c, \hat{e}(g,g)^{y}) = 1] \leq \mu(\lambda)$$

where $a, b, c, y \in \mathsf{Z}_p$ and $\mu(\lambda)$ is a negligible over λ;

Thus, DBDH holds if, there is no probabilistic polynomial-time adversary can distinguish $[g^a, g^b, g^c, \hat{e}(g,g)^{abc}]$ from $[g^a, g^b, g^c, \hat{e}(g,g)^{z}]$ with non-negligible advantage.

2.14.3 Computational Diffie-Hellman Assumption

Definition 2.14 (Computational Diffie-Hellman Assumption)**.** *Let λ be a security parameter and $(p, \mathsf{G}, \mathsf{G}_T, \hat{e}, g)$ be a uniformly randomly generated tuple of bilinear pairing parameters from $\mathtt{BMGen}(1^\lambda)$. It is said to be hold* computational Diffie-Hellman assumption *if, given g^a, g^b where $a, b \xleftarrow{\$} \mathsf{Z}_p^*$, any probabilistic polynomial time adversary $\mathcal{A}$ can compute $g^{ab} \in \mathsf{G}$ only with negligible probability, namely*

$$Adv_{\mathcal{A}}^{CDH}(\lambda) = \Pr_{a,b \xleftarrow{\$} \mathsf{Z}_p}[\mathcal{A}(g, g^a, g^b) = g^{ab}] \leq \mu(\lambda)$$

i.e., no PPT algorithm $\mathcal{A}$ can output $g^{ab} \in \mathsf{G}$ with more than a negligible advantage.

2.14.4 q-Strong Diffie-Hellman Assumption

Definition 2.15 (q-Strong Diffie-Hellman Assumption)**.** *[76]. Let λ be a security parameter and $(p, \mathsf{G}, \mathsf{G}_T, \hat{e}, g)$ be a uniformly randomly generated tuple of bilinear pairing parameters from $\mathtt{BMGen}(1^\lambda)$. Given an upper bound q, an element $s \xleftarrow{\$} \mathsf{Z}_p^*$ and the set $\{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$, it is said to be hold* q-strong Diffie-Hellman *$(q$-SDH$)$ assumption if, any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ can find a pair $(c, g^{\frac{1}{x+c}})$ only with negligible probability, namely*

$$Adv_{\mathcal{A}}^{q\text{-}SDH}(\lambda) = \Pr\left[\mathcal{A}(g, g^s, g^{s^2}, \cdots, g^{s^q}) \to (s, g^{\frac{1}{s+c}})\right] \leq neg(\lambda),$$

where $c \in \mathsf{Z}_p$.

2.15 Signature Based on GDH Groups

Boneh et al. [13] first presented a signature scheme based on bilinear map over GDH Group. Let $\mathcal{H} : \{0,1\}^* \to \mathsf{G} \setminus \{1\}$ be a full-domain one-way hash function. The security analysis views $\mathcal{H}$ as a random oracle. The scheme can be described as follows.

Definition 2.16 (BLS signature scheme)**.** *Let $\hat{e} : \mathsf{G} \times \mathsf{G} \to \mathsf{G_T}$ be a bilinear map where $|\mathsf{G}| = |\mathsf{G_T}| = \mathsf{p}$, a prime and $\mathsf{G} =< g >$. A* BLS *signature scheme $\mathcal{S}$=($\mathtt{KeyGen}$, $\mathtt{Sign}$, $\mathtt{Verify}$) is given as a tuple of three algorithms as follows.*

- $(sk, pk) \leftarrow \mathtt{KeyGen}$*: It selects $\alpha \xleftarrow{\$} \mathsf{Z_p}$. It keeps the private key $sk = \alpha$ and publishes the public key $pk = g^\alpha$.*

- $\sigma \leftarrow \texttt{Sign}(sk, m)$*: Given* $sk = \alpha$*, and some message* m*, it outputs the signature* $\sigma = (\mathcal{H}(m))^{\alpha}$.

- $\{0/1\} \leftarrow \texttt{Verify}(pk, m, \sigma)$*: For a message* m*, signature* σ *with public key* pk*, it check whether* $g, pk, \mathcal{H}, \sigma$ *is a Diffie-Hellman tuple by verifying equality between* $\hat{e}(\sigma, g)$ *and* $\hat{e}(\mathcal{H}(m), pk)$*, where* $\hat{e} : \mathsf{G} \times \mathsf{G} \rightarrow \mathsf{G_T}$ *is a bilinear map.*

2.15.1 Security of BLS signature scheme

Definition 2.17 (Secure BLS signature scheme)**.** *Let* $\mathcal{S}$=(KeyGen, Sign, Verify) *be a bilinear signature scheme as described above. Let us consider the* existential unforgeability under a chosen-message attack *game* $Sig_{\mathcal{A}}$ *between a challenger and an adversary* $\mathcal{A}$ *as follows.*

1. *The challenger runs* $(sk, pk) \leftarrow \texttt{KeyGen}$*.* $\mathcal{A}$ *is given* pk*.*
2. $\mathcal{A}$ *requests signature for* q_s *messages* $m_1, m_2, \ldots, m_{q_s} \in \{0,1\}^*$*, chosen adaptively, and gets responses* $\sigma_1, \sigma_2, \ldots, \sigma_{q_s}$*.*
3. *Eventually,* $\mathcal{A}$ *outputs* (m, σ) *and wins the game if (1)* $m \notin \{m_1, m_2, \ldots, m_{q_s}\}$*, and (2)* $\texttt{Verify}(pk, m, \sigma)$ = *valid.*

We define $AdvSig_{\mathcal{A}}^{\mathcal{S}}$ *to be the probability that A wins in the above game, taken over the coin tosses of* KeyGen *and of* $\mathcal{A}$*.*

Let, a forger $\mathcal{A}(t, q_s, q_H, \epsilon)$ *runs in time at most t, makes at most* q_s *signature queries, at most* q_H *hash queries.* $\mathcal{S}$ *is said to be* secure *from existential unforgeability under a chosen-message attack, if for all such adversary* $\mathcal{A}$*,* $AdvSig_{\mathcal{A}}^{\mathcal{S}} < \epsilon$

Theorem 2.1. *Let* G *be a* (τ, t', ϵ')*-GDH group of prime order* p*. Then the BLS signature scheme* $\mathcal{S}$ = (KeyGen, Sign, Verify) *is* (t, q_s, q_H, ϵ)*-secure against existential forgery under an adaptive chosen-message attack in the random oracle model,* $\forall t, \epsilon$ *satisfying*

$$\epsilon \geq e(q_s + 1).\epsilon' \text{ and } t < t' - c_{\mathsf{G}}(q_H + 2q_s)$$

where, c_{G} *is a constant that depends on* G*, and* e *is the base of the natural logarithm.*

Proof of the above theorem is given in [13].

2.16 BGN Encryption Scheme

Boneh *et al.* [12] proposed a homomorphic encryption scheme (henceforth referred to as BGN encryption scheme) that allows an arbitrary number of additions and one multiplication. Before presenting the scheme we first briefly describe the subgroup decision problem as follows.

2.16.1 The subgroup decision problem

Definition 2.18 (subgroup decision assumption)**.** *[12]. $\mathcal{G}$ is a polynomial algorithm which, given a security parameter $\lambda \in \mathsf{Z}^+$ as input, outputs as follows.*

1. *Generate two random $\lambda - bit$ primes q_1, q_2 and sets $n = q_1 q_2 \in \mathsf{Z}$.*
2. *Generate $(n, \mathsf{G}, \mathsf{G}_1, g, \hat{e}) \leftarrow \texttt{BMGen}$ as described in Section 2.13, where G is a bilinear group of order n, $\hat{e} : \mathsf{G} \times \mathsf{G} \rightarrow \mathsf{G}_1$ is a biliear map.*
3. *Finally, it outputs $(q_1, q_2, \mathsf{G}, \mathsf{G}_1, e)$*

Now given $(n, \mathsf{G}, \mathsf{G}_1, \hat{e})$, without knowing the factorization of n, the subgroup decision problem is to decide if an element $x \in \mathsf{G}$ is in a subgroup of G. We say $\mathcal{G}$ satisfies subgroup decision assumption when the uniform distribution on G is indistinguishable from the uniform distribution on a subgroup of G .

2.16.2 The scheme description

A BGN encryption scheme consists of three algorithms- $\texttt{Gen}()$, $\texttt{Encrypt}()$ and $\texttt{Decrypt}()$.

Algorithm 1: $\texttt{Gen}(1^\lambda)$

$(q_1, q_2, \mathsf{G}, \mathsf{G}_1, e) \leftarrow \mathcal{G}(\lambda)$
$n \leftarrow q_1 q_2$
$g \xleftarrow{\$} \mathsf{G}$; $r \xleftarrow{\$} [n]$
$u \leftarrow g^r$; $h \leftarrow u^{q_2}$
$sk \leftarrow q_1$; $pk \leftarrow (n, \mathsf{G}, \mathsf{G}_1, e, g, h)$
return (pk, sk)

Key generation: This takes a security parameter λ as input and outputs a public-private key pair (pk, sk) (see Algo. 1). Here, $pk = (n,\ \mathsf{G},\ \mathsf{G}_1,\ e,\ g,\ h)$ and $sk = q_1$. In pk, e is a bilinear map

from $\mathsf{G} \times \mathsf{G}$ to G_1 where both G and G_1 are groups of order q_1. Note that, given λ, $\mathcal{G}$ returns $(q_1,\ q_2,\ \mathsf{G},\ \mathsf{G}_1,\ e)$ (see [12]) where q_1 and q_2 are two large primes, and G and G_1 are groups of order $n = q_1 q_2$.

Algorithm 2: $\mathtt{Encrypt}_{\mathsf{G}}(pk, a)$

$(n, \mathsf{G}, \mathsf{G}_1, e, g, h) \leftarrow pk$
$r \xleftarrow{\$} [n]$
$c \leftarrow g^a h^r$
return c

Algorithm 3: $\mathtt{Decrypt}_{\mathsf{G}}(pk, sk, c)$

$(n,\ \mathsf{G},\ \mathsf{G}_1,\ e,\ g,\ h) \leftarrow pk$; $q_1 \leftarrow sk$
$c' \leftarrow c^{q_1}$; $\hat{g} = g^{q_1}$
$s = D\log_{\hat{g}} c'$
return s

Encryption: An integer a is encrypted in G using Algo. 2. Let a_1 and a_2 be two integers that are encrypted in $\mathcal{G}$ as c_1 and c_2. Then, the bilinear map $e(c_1, c_2)$, belongs to G_1, gives the encryption of $(a_1 a_2)$. Note that arbitrary addition of plaintext is also possible in the group G_1. If g is a generator of the group G, $e(g, g)$ acts as a generator of the group G_1. Thus, the encryption of an integer a is possible in G_1 in similar manner (see Algo. 4).

Algorithm 4: $\mathtt{Encrypt}_{\mathsf{G}_1}(pk, a)$

$(n, \mathsf{G}, \mathsf{G}_1, e, g, h) \leftarrow pk$
$r \xleftarrow{\$} [n]$
$g_1 \leftarrow e(g, g)$; $h_1 \leftarrow e(g, h)$
$c \leftarrow (g_1)^a (h_1)^r$
return c

Algorithm 5: $\mathtt{Decrypt}_{\mathsf{G}_1}(pk, sk, c)$

$(n, \mathsf{G}, \mathsf{G}_1, e, g, h) \leftarrow pk$
$q_1 \leftarrow sk$
$c' \leftarrow c^{q_1}$; $\hat{g}_1 = e(g, g)^{q_1}$
$s = D\log_{\hat{g}} c'$
return s

Decryption: At the time of encryption each entry is randomized. The secret key q_1 eliminates the randomization. Then, it is enough to find discrete logarithm $D\log$ of the rest. Algo. 3 and Algo. 5 describes the decryption in G and G_1 respectively. In decryption algorithms, $D\log$ computation can be done with expected time $O(\sqrt{n})$ using Pollard's lambda method [64]. However, it can be done in constant time using some extra storage ([12]).

Let `BGN` be an encryption scheme as described above. Then, it is a tuple of five algorithms (`Gen`, $\mathtt{Encrypt}_{\mathsf{G}}$, $\mathtt{Decrypt}_{\mathsf{G}}$, $\mathtt{Encrypt}_{\mathsf{G}_1}$, $\mathtt{Decrypt}_{\mathsf{G}_1}$) as described in Algo. 1, 2, 3, 4 and 5 respectively.

2.16.3 Security of the BGN Encryption Scheme

The security of the `BGN` encryption scheme lies on the subgroup decision assumption (see Definition 2.18) of $\mathcal{G}$. For the security of `BGN` scheme, we have the following theorem.

Theorem 2.2. *The public key system* BGN *is semantically secure assuming $\mathcal{G}$ satisfies the subgroup decision assumption.*

2.17 Dynamic universal accumulator

Cryptographic accumulators are used to prove membership/non-membership of elements in a set. A client uses accumulator when it wants to outsource the set to a third party cloud server but keeping the ability of membership query. When the size of the set is large, proof generation and (or) proof size becomes expensive. Accumulation tree enable a client to outsource large set in an efficient manner. Though the existing accumulator scheme like [76] can build accumulation tree for static database which can provide the proof of membership as well as non-membership, it is efficient for dynamic set. Au et al. [5] presented a scheme that dealt with dynamic set that generates membership proofs efficiently. They extended their scheme with an additional authenticated tree that allows non-membership check. However, this structure does not support update. (Verify)

A dynamic universal accumulator (DUA) allows one to outsource a set of elements with ability to query the existence of an element together with a functionality to verify the result and way to update the set.

There are two kinds of widely used accumulators– RSA accumulator and bilinear map accumulator. The binlinear map accumulators generates shorter proof of membership compare to RSA accumulators. In our case, we take bilinear map accumulators.

Let us consider a DUA proposed by Au et al. [5]. Let $\mathtt{AC} = (\mathtt{Init}, \mathtt{Gen}, \mathtt{Update}, \mathtt{MemWitGen}, \mathtt{MemWitVer})$ be such a DUA described as follows.

Initialization. $(s, tup) \leftarrow \mathtt{AC.Init}(\lambda)$:
Given a security parameter λ, let us consider a uniformly generated tuple $tup = (p, \mathsf{G}, \mathsf{G}_T, g, \hat{e})$ of bilinear pairing parameters generated with BMGGen. Then $\hat{e} : \mathsf{G} \times \mathsf{G} \rightarrow \mathsf{G}_T$ be a bilinear pairing such that $|\mathsf{G}| = |\mathsf{G}_T| = p$ for some λ-bit prime p and $\mathsf{G} =< g >$. Let q be the maximum number of elements to be accumulated. Then a uniformly random element $s \xleftarrow{\$} Z_p^*$ is selected. s is treated as secret key.

Accumulator Generation. $Acc(Y) \leftarrow \mathtt{AC.Gen}(Y, s)$:
Given a set $Y = \{y_1, \ldots, y_k\} \in Z_p^*$. Let $v(s) = \prod_{y \in Y}(y + s) \pmod p$ be a polynomial of degree $k \leq q$. Then the accumulator is $Acc(Y) = g^{v(s)}$, which can be computed efficiently.

Updating accumulator $Acc(Y') \leftarrow \mathtt{AC.Update}(Acc(Y), s, \bar{y}, op =$ add or delete): Let $Acc(Y)$

be the accumulator value for a set of elements $Y = \{y_1, \ldots, y_k\} \in \mathsf{G}_p$. If an element $\bar{y}$ is added, the new accumulator value will be $Acc(Y') = Acc(Y)^{\bar{y}+s}$, where $Y' = Y \cup \{\bar{y}\}$. Similarly, if an element $\bar{y}$ is deleted, the accumulator changes to $Acc(Y') = Acc(Y)^{\frac{1}{\bar{y}+s}}$, where $Y' = Y \setminus \{\bar{y}\}$. For both case, the secret value s is needed to compute updated value.

Membership witness generation $wt(\bar{y}) \leftarrow \mathtt{AC.MemWitGen}(PG, Y, \bar{y})$:
For a set of elements $Y = \{y_1, \ldots, y_k\} \in Z_p^*$, a membership witness $wt(\bar{y})$ for the element $\bar{y} \in Y$ is given by $wt(\bar{y}) = \left[g^{\prod_{i=1}^{k}(y_i+s)}\right]^{\frac{1}{\bar{y}+s}} = [Acc(Y)]^{\frac{1}{\bar{y}+s}}$. This witness also can be computed without the knowledge of s as follows. Expanding the polynomial $\prod_{i=1, i\neq j}^{k}(y_i + s)$, it can be written as $\sum_{i=0}^{i=k-1} u_i s^i$. Then, the witness is computed as $wt(\bar{y}) = g^{\prod_{i=1,i\neq j}^{k}(y_i+s)} = g^{\sum_{i=0}^{i=k-1} u_i s^i} = \prod_{i=0}^{i=k-1} g^{s^i u_i} = \prod_{i=0}^{i=k-1} g_i^{u_i} \in \mathsf{G}$.

Since, in our scheme, only the client, who has the key s, generates witnesses, we only consider first method.

Membership witness verification $b_v \leftarrow \mathtt{AC.MemWitVer}(Acc(Y), \bar{y}, wt(\bar{y}))$:
Membership witness is verified by checking whether $\hat{e}(Acc(Y), g) = \hat{e}(wt(\bar{y}), g^{\bar{y}+s})$. Finally, a verification bit b_v is returned where $b_v = 1$ if equality holds and $b_v = 0$ otherwise.

The correctness of membership verification is given as follows.

$$\hat{e}(wt(\bar{y}), g^{\bar{y}+s}) = \hat{e}(g^{\prod_{y\in Y, y\neq\bar{y}}(y+s)}, g^{\bar{y}+s}) = \hat{e}(g^{\prod_{y\in Y}(y+s)}, g) = \hat{e}(Acc(Y), g)$$

Non-membership witness $\mathtt{AccNonMemWitGen}(PG, Y, \bar{y})$.
For a set of elements $Y = \{y_1, \ldots, y_k\} \in \mathsf{G}_p$, a non-membership witness $wtn(\bar{y}) = (c, d)$, for some $\bar{y} \notin Y$, can be computed without the knowledge of s as well, where $d = \prod_{i=1}^{k}(y_i + s) \pmod{()\bar{y} + s}$ and $c = g^{\frac{\prod_{i=1}^{k}(y_i+s)-d}{\bar{y}+s}} \in \mathsf{G}$.

Let $v(s) = \prod_{i=1}^{k}(y_i + s)$. Since, $\bar{y} \neq y_i$ for all i, there exists a degree $k-1$ polynomial $c(s)$ and a constant d such that $v(s) = c(s)(s + \bar{y}) + d$. Expand c and write it as $c(s) = \sum_{i=0}^{i=k-1}(u_i s^i)$. Therefore $c = g^{c(s)} = g^{\sum_{i=0}^{i=k-1}(u_i s^i)} = \prod_{i=0}^{k-1} g^{u_i s^i} = \prod_{i=0}^{k-1} g_i^{u_i}$

Non-membership witness verification
$\mathtt{AccNonMemWitVer}(Acc(Y), wtn(\bar{y}) = (c, d), \bar{y})$.

Membership witness can be verified by checking the equality between $\hat{e}(Acc(Y), g^{-d})$ and $\hat{e}(c, g^{\bar{y}+s})$. The correctness of the verification can be given as

$$\begin{aligned}
\hat{e}(c, g^{\bar{y}+s}) &= \hat{e}(g^{c(s)}, g^{\bar{y}+s}) \\
&= \hat{e}(g^{c(s)(\bar{y}+s)}, g) \\
&= \hat{e}(g^{v(s)-d}, g) \\
&= \hat{e}(g^{v(s)}, g^{-d}) \\
&= \hat{e}(Acc(Y), g^{-d})
\end{aligned}$$

Security of the accumulator The security of the above construction lies on the strong Diffie-Hellman assumption given in Definition 2.15. See [5] for the details.

Theorem 2.3. *Let λ be a security parameter and $(p, \mathsf{G}, \mathsf{G}_T, \hat{e}, g)$ be a uniformly randomly generated tuple of bilinear pairing parameters. Given an upper bound q and the set $S = \{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$, for some randomly chosen s from Z_p^*, the probability a PPT adversary $\mathcal{A}$ can find another set $S' \neq S$ ($|S'|$) such that $Acc(S) = Acc(S')$ is $neg(\lambda)$.*

See [76] for the detail proof.

2.18 Garbled Circuit (GC)

Let us consider two parties, with input x and y respectively, who want to compute a function $f(x, y)$. Then, a garbled circuit [105, 59] allows them to compute $f(x, y)$ in such a way that none of the parties get any 'meaningful information' about the input of the other party and none, other than the two parties, is able to compute $f(x, y)$. However, every function can be converted into a Boolean circuit. A Garbled circuit is a garbled version of the Boolean circuit.

We can briefly summarize the main ideas of the protocol of a garbled circuit as follows. A garble circuit consists of two algorithms `CreatGC` and `EvalGC`. There are a constructor and an evaluator, in general, they are cloud server and client respectively. The constructor creates a garbled circuit GC of a circuit using the algorithm `CreatGC`. For each wire W_i of the circuit, it chooses two garbled values w_i^0 and w_i^i randomly and selects one of them at random as garble value of W_i.

In the next phase, corresponding to each gate G_i, the constructor creates a garbled table T_i. Given a set of garbled input values corresponding to G_i, evaluator is only allowed to recover nothing but the garbled value of the corresponding G_i 's output from T_i. After the construction,

the constructor sends the garbled tables to the evaluator. Thereafter, the evaluator obliviously gets the garbled inputs corresponding to inputs of both parties.

Then, using the algorithm `EvalGC`, it evaluates the garbled circuit on the garbled inputs. Using the garbled tables T_i, it obtains the garbled outputs simply by evaluating the garbled circuit gate by gate, . Finally, it gets output values given for the respective players by transforming the garbled outputs.

Kolesnikov *et al.* [52] introduced an optimization of garbled circuit that allows XOR gates to be computed without communication or cryptographic operations [97]. Kolesnikov *et al.* [51] presented efficient GC constructions for several basic functions using the garbled circuit construction of [52]. In this paper, we use garbled circuit blocks for subtraction (`SUB`), comparison (`COMP`) and multiplexer (`MUX`) functions from [52].

Chapter 3

Literature Survey

In this chapter of the thesis, at first we categorize queryable encryption schemes with different aspects. Then we discuss a short survey of existing literature. Rest of the chapter is organized as follows.

In Section 3.1, we discuss the literature on queryable encryption over text data i.e., searchable encryption. At first, we briefly describe schemes over static data in Section 3.1.1. Then in Section 3.1.2, we move to dynamic searchable encryption schemes that support updates. After that, in Section 3.1.3, we discuss some important attacks over those schemes. Then in Section 3.1.4 we go into forward and backward private schemes that are secured from those attacks. Thereafter, in Section 3.1.5, we talk about verifiable schemes that are resilient even in presence of malicious adversaries Finally, in Section 3.2, we discuss queryable encryption schemes over graph data.

3.1 Queryable Encryption on text data

We see that Queryable encryption is called Searchable Encryption when the data is text data. In this section, we describe some of the important works on searchable encryption.

3.1.1 Searchable Symmetric Encryption (SSE)

Song et al. [86] first introduce a database encryption method where a keyword search query can be performed. The scheme is for static databases and is provably secure. The scheme allows the cloud servers to search only for the authorized words i.e., for a fixed dictionary. Before giving the final scheme, they proposed three more techniques one by one, each is an improvement of the previous. In all schemes, each document was encrypted separately. The encryption of a document was individual encryption of the words in it. When a client needs to search a document, it sends the cloud server the corresponding information so that the cloud server gets the encrypted words in the documents and returns the results if present.

The designed schemes are quite simple and easy to compute. However, in all of them, during a search, *every keyword of every file has to be decrypted*. This is decrypting the entire database for

each search query. When the number of documents is high then the scheme becomes inefficient.

In the technique [86] to search on encrypted data, the owner encrypts each file keyword by keyword. Instead, Eu-Jin Goh [38], uses a secure index which is an encrypted index make specifically for searching. The actual documents are encrypted differently and kept separately. The search index keeps an identifier of the documents. While searching, the search index returns the set of identifiers, and then the cloud returns the documents. He defines secure index formally and formulates a security model for the indexes.

Later, he presents a search scheme based on the bloom filter. Briefly say, he keeps a bloom filter for every document and uses multiple hash functions that map each document-keyword pair to the bloom filter. While searching for a keyword, the cloud visits the bloom filter corresponding to each document and then searches if all the hashes match. If it matches all, then the cloud returns the document in the result set. Though the scheme looks very fast as it uses only symmetric key encryption scheme and hash functions, the search cost linearly grows with the number of documents. So, it becomes inefficient when the number of documents is large. Moreover, the scheme was only for static data.

Chang and Mitzenmacher [26] design a scheme for keyword search on encrypted data. They consider the clients to be lightweight like a mobile device. In their scheme, they do not encrypt each file separately. Instead, they consider a bit-string of the length of the dictionary for each file where an entry bit is 1 if the corresponding keyword is present. Then each bit is masked with a pseudo-random bit generated with a pseudo-random generator. While searching for a keyword, it reveals all corresponding positions in all strings. So, in the scheme, the search complexity is linear in the number of documents which is highly inefficient when the number of documents is large.

Curtmola et al. [32] first introduce the notion of searchable encryption for single keyword search. They formally define it and show that they are provably secure. Instead of encrypting each file separately as in [86, 26], they use inverted-index. For each keyword, the set of the file identifiers containing the keyword is kept in a linked list. Then they encrypt the link list and search over it when required. In doing so, the computational complexity for searching is reduced to the order of size of search results which is also a huge reduction. The communication complexity for searching also is reduced to only one round. Moreover, they present the leakage of a searchable scheme formally. Though they used only symmetric key encryption techniques in their scheme, the scheme is only for static databases.

3.1.2 Dynamic Searchable Symmetric Encryption (DSSE)

Inverted index based schemes Searching over encrypted data with updates is first considered by Kamara et al. [48]. The scheme is an improvement over Curtmola et al.[32]. It is an inverted index-based searchable encryption and uses two indices– an inverted index and a general index. The inverted index is used for search and addition while the general index is used for deleting documents. Both indices are linked with each other. In the inverted index, corresponding to each keyword, there is a linked list that stores the identifiers of the documents containing the keyword. In the general index, corresponding to each document, there is a linked list that stores addresses of the containing keywords. Though the second type of index is not required during a search, an update operation requires both of them to be modified.

This scheme also uses only symmetric encryption schemes, and they claim that the scheme has optimal search time. Their implementation result also ensures its good performance. However, since each node of the linked lists is taken from a random location of a large array, the search or update operation becomes sequential.

Red-black tree based dynamic schemes We see, previously in [48], the search or the update operation was sequential. Kamara and Papamanthou [47] try to improve that. They use a tree-based multi-map data structure, called keyword red-black tree (KRB tree), and propose a new single keyword search scheme where the search or update operations can be done with parallel computation.

KRB Tree is a m-ary tree where each leaf node corresponds to a document. Briefly speaking, the internal nodes are the path to the documents and store information about the keywords that are contained in the documents below them. To keep all the keywords, each internal node contains a bit-string of the size of keyword-dictionary. While adding a document, all internal nodes in the path to the document keep the keywords, that are not already there. In a similar way, the deletion can be done.

Their scheme has an extra advantage in leakage. In the scheme, the client does not leak information about the keywords that are contained in a newly updated document.

Oblivious Cross-Tags (OXT) protocol Cash et al. [22] propose an SE scheme that supports conjunctive as well as Boolean queries. They present the concept of T-Set that allows one to associate a list of fixed-sized data tuples with each keyword in the database. Later, it enable them to issue keyword-related tokens to retrieve those lists. Since T-Set is similar to encrypted inverted

index, it can be used as a single keyword search SE scheme, Later they propose two protocols– Basic Cross-Tags (BXT) protocol and Oblivious Cross-Tags (OXT) protocol.

In BXT protocol, For each $w \in W$, a value $xtrap = F(K_X, w)$ is computed where K_X is a PRF key. Then for each $ind \in DB(w)$ a value $xtag = f(xtrap, ind)$ is computed and added to $XSet$ where f can be a PRF. During search for $(w_1, \ldots, w_n)$, the protocol chooses the estimated least frequent keyword, say w_1 , finds single keyword search w_1, say the result is R_{w_1}. Then for each files ind in R_{w_1}, it recomputes $xtag = f(xtrap, ind)$, where $xtag$s are given in search token. For a file if all such $xtag$ exists in corresponding $TSet$ then the file is included in the final result.

We see that since $xtrap$s are given for the queried keywords, the cloud server can store them. Later when some document ind is revealed, the cloud server can check whether the keyword corresponding to the $xtrap$ is present in file ind by simply checking the presence of $f(xtrap, ind)$ in $XSet$. Note that, BXT reveals the keys from which the cloud server able to compute $f(xtrap, .)$ itself. This makes the attack possible.

In OXT protocol xtags are computed differently as $xtag = g^{xtrap.xind}$ which is function of $xtrap$ and $xind$ instead of $xtrap$ and ind. Where, $xind$ is encrypted ind. To decrypt $xind$, an extra information $y = xind.z^{-1}$ is kept in the tuple set together with another encrypted version e of $xind$, where $z = F_p(K_Z, w||c)$ and c is a counter. During search, client recomputes $xtoken = g^{xtrap.F_p(K_Z,w||c)}$ sends them to the cloud. The cloud matches the $xtoken^y$ with $xtag$. Since the client does not store the counters, it sends continuously until matched. So, the number of interactions becomes unbounded here.

Later, they show that their OXT scheme is semantically secure against adaptive adversarial attacks. They present a prototype of their scheme, run it on several large real-world data sets and show the performance. Moreover, the scheme is first to give conjunctive search results in sublinear time.

Keyword search scheme using blind storage Naveed et al. [72] propose the concept of blind storage which is a scheme that allows a client to outsource a set of documents in such a way that the individual size and the number of the documents remains hidden from the cloud. A blind storage is an array of blocks where the cloud only can update and returns blocks requested by the client. The cloud does not need any computation.

The basic idea of the scheme is as follows. Like inverted index, the client prepares a list of documents which is then divided into a set of blocks which are then mapped to random locations in the blind storage blocks. While searching for a keyword, the corresponding blocks are accessed

together with some extra noise blocks. The concept of these noise blocks hides the size of the actual size of the data. The client rearranges the blocks and gets the actual result.

Though in the scheme the leakage is reduced and is satisfies a fully adaptive security model, the client has to perform all computational work. The cloud becomes a storage service provider. This increases the computational cost of the client a lot. So the scheme is not suitable for lightweight clients.

Parallelizable keyword search Cash *et al.* [21] introduced first scheme that allows parallelism in input/output in large scale. The parallelism enables the system to work with ten billion keyword file pairs. The scheme doesn't claim space after deletion though it is an append-only scheme. However, they consider only the cases where deletions are rare. Moreover, they consider the cloud server cannot add duplicate file-identifiers or delete currently non-existing file identifiers, or delete a keyword from a file identifier that doesn't match it.

In their scheme, as an add query, instead of a keyword-document pair, a file as a whole is added and this prevents it to modify a document. In addition, in their proposed dynamic scheme Π_{bas}^{dyn}, either the client has to store the dictionary of size $|\mathcal{W}|$ or it can outsource in encrypted form. In the latter case, the client has to download the dictionary in each session. This causes an extra round of communication. Though the scheme is dynamic, it is not forward or backward secure.

History-based keyword search At a similar time, Hahn and Kerschbaum [42] proposed a history-based dynamic searchable encryption scheme. The scheme can be briefly described as follow.

Initially, for each document f, the client extracts the set of keywords $w_1, w_2, \ldots, w_{|f|}$. Then for each keyword w_i a random bit string $c_i = H_{\tau_{w_i}}(s_i)||s_i$ is generated where H is a hash function, $\tau_{w_i} = F_{k_1}(w_i)$ and the s_is are generated with a pseudo-random number generator and fixed for all documents. Finally, all such strings c_i's are stored in a list corresponding to the file.

During search, given τ_{w_i} the cloud searches all $c_i = l_i||r_i$ and checks if $l_i = H_{\tau_{w_i}}(r_i)$ and makes a list of corresponding document identifiers. This is done only when a keyword is searched the first time. Thereafter the list is stored as history either. The list is either stored at client-side or cloud-side. Keeping the searched history stored saved the same further computational cost can be avoided. During adding a new document or deleting an existing one, the cloud checks the containing keywords are already searched. If so, the cloud edits historical lists corresponding to the keywords.

Though the initial search time is linear in the number of document-keyword pairs, they show

that it amortizes over multiple searches and a theoretic upper bound for amortization in $O((|\mathcal{W}|)^2)$ where W is the set of keywords. However, the concept of keeping the previous search history stored on the cloud side is applicable effective for any queryable encryption scheme at the cost of extra cloud storage.

Boolean search with worst-case sub-linear complexity The keyword search scheme by Cash et al. [22] has sublinear search time for conjunctive queries and linear search time for the arbitrary disjunctive and Boolean queries. Improving the result, Kamara et al. [46] present a scheme that has worst-case sublinear search complexity disjunctive and Boolean keyword search in presence of optimal communication complexity.

They presented three schemes–IEX, ZMF, and DIEX. A single keyword SE scheme is used, in the main construction IEX, as a black-box multi-map. There are an encrypted *global* multi-map and an encrypted dictionary in the construction. Those maps every keyword w, respectively, to its document identifiers $DB(w)$, and to a *local* multi-map for w. All the keywords v that co-occur with a keyword w are mapped, by the local multi-map of w, to the identifiers of the documents that contain both v and w. During disjunctive search for $\{w_1, w_2, \ldots, w_n\}$, $DB(w_1)$ can be recovered from encrypted global multi-map and then w_2 to w_n encrypted local multi-map can be recovered. Finally, from local multi-map the common identifiers can be found. These common identifiers help to find disjunctive query results in worst-case sub-linear time. For the Boolean query, before processing, the query is converted in CNF form. Then first inner disjunctions are found, and then outer conjunctions are found.

The second scheme ZMF relies on a new single keyword SSE scheme and is adaptively secure. It is inspired from [38] and produces a collection variable-sized encrypted Bloom filters, called Matryoshka filter, which are adaptively secure. In the scheme, local multi-maps are replaced with a Matryoshka filter. It is instantiated in different ways. The first instantiation IEX-2Lev uses [87] as black-box. At the cost of storage overhead, is optimized for search. The second instantiation IEX-ZMF uses their own construction ZMF. At the cost of efficiency, it is optimized for storage, and still achieves asymptotically sub-linear search.

The third scheme, DIEX is the dynamic version of IEX. It supports sub-linear time Boolean search queries.

Outsourcing query to blockchain Adkins et al. [2] discuss three schemes LSX, TRX, PAX. This is not a typical searchable encryption scheme. Instead, they provide secure databases for keeping key-value pairs i.e., label-value pairs. The value is a single value for dictionary and set of

values i.e., tuples.

The first scheme stores values in a tuple one by one in the BC database. Each value stores the address of the previous value. The final value is stored on the client-side. These form a link list. The difference between LSX & [32] is that in [32] address of the nodes are controlled by the client, but in the LAX by the public ledger. In TRX, a virtual tree structure is considered for blocks. Each layer, starting from leaves, is uploaded at a time and their addresses are kept in the next upper layer. The root address is stored on the client-side.

LSX & TRX both are append-only schemes. The blockchain database itself does not support deletion. So, in PAX, we skip deleted entries. The required information is kept in a patch dictionary. PAX is stored with LSX with an additional data structure.

Finally, the scheme only gives storage for multi-maps. So, searchable encryption schemes are done only client-side in blocks. So we think, the blockchain database is not required to do so. Any cloud service provider can be used. They did not show any application except storing multi-maps

Query in presence of multiple clients Xu et al. [103] proposed a DSE scheme for medical databases. They considered a multi-client model in presence of a trusted central authority that manages access control of the data among clients. There is a cloud service provider that keeps encrypted data as well as encrypted search index and performs search and updates whenever they are needed.

The idea of the scheme is as follows. In the beginning, each client collects secret keys (sk_S, sk_w) corresponding to its attributes S and the set of authorized keywords respectively. The keys are used for searching.

During building, for a keyword w, a file identifier id is encrypted with an HBE scheme and again encrypted with some random string fixed for that pair (w, id). Then it is kept in a table at a position that depends on the pair (w, id)

During a search, given an authorized keyword w, the cloud server first finds the positions, in the table, of the identifiers belonging to $DB(W)$. Then decrypts them and sends them to the client. In this set, all identifiers are encrypted with attributes. The client can decrypt them if it has a satisfying set of attributes. Later, from the result set of identifiers, the client can request the files that it wants.

However, the set $\mathcal{W}$ of keywords need to be authorized by the central authority beforehand that prevents free searching over the database. When the size of the set increases, the time to generate token increases for the client. Besides, the scheme is a single keyword search scheme only. Since

it is a result hiding scheme, it requires two rounds of communication to get the search result.

Top-k nearest keyword search queries A queryable encryption scheme, that supports top-k nearest keyword search queries, has been proposed by Liu *et al.* [60]. They have made an encrypted index using order-preserving encryption for searching. Together with lightweight symmetric key encryption schemes, homomorphic encryption is used to compute over encrypted data.

In the 'top-k nearest keyword' query supporting scheme, Liu *et al.* [60], have used order-preserving encryption that contains homomorphic encryption.

3.1.3 Attacks on Queryable Encryption over text data

Besides different SE schemes, few papers on the attacks on SSE schemes have been published since the last decade. We describe them briefly as follows.

Access pattern disclosure Islam et al. [43] presented the first paper on the attack on Searchable Encryption (SE). They investigated different types of attacks due to access pattern disclosure in SE schemes, formalized a *query identity inference attack* model based on access pattern disclosure, and proposed a noise addition technique to mitigate such an attack. In query identity inference attacks, by combining leakage with publicly-available information [71], the adversary tries to recover information about the data or queries.

The attack assumes the adversary has access to some prior knowledge on the document set. The adversary has full access to the communication channel and observes a sequence of l queries $\mathcal{Q} = \mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_l$ and sequence of their responses $R_{\mathcal{Q}} = R_{\mathcal{Q}_1}, R_{\mathcal{Q}_2}, \ldots, R_{\mathcal{Q}_l}$. Secondly, it knows the underlying keywords for k of the queries in the sequence, and the $m \times m$ co-occurrence matrix where m is the number of possible keywords and (i, j)th entry is the probability of co-occurrence of ith and jth keywords. The attack is based on frequency analysis that shows the attack is possible even when $k \ll l$. Finally, to prevent this attack they propose to add false-positive results in the pre-processing stage. Adding false positives makes it difficult to the adversary to compare the frequencies between the co-occurrence matrix and the query results.

Inference attacks Naveed et al. [71] studies inference attacks. They propose frequency analysis attack and l_2-optimization attack for deterministic encryption (DTE) encryption schemes and sorting attack and cumulative attack for order-preserving encryption (OPE) schemes. The attacks lie on certain assumptions i.e., prior knowledge of the database.

Frequency analysis attack is traditional, since DTE schemes have a bijection from the keywords to its encrypted version, frequencies can be compared. This type of attack is applicable for schemes like [86] only if the distribution of plain text keywords are is known to the adversary. l_2-optimization attack is similar to the frequency analysis. However, it uses the l_p optimization problem with a histogram to match frequencies.

In sorting attack, the order can be computed over an OPE-encrypted dense column **c** over cipher-text space C_k. So, the adversary simply sorts c and plain-text space M_k. Then it outputs a function that maps each ciphertext $c \in \mathbf{c}$ to the element of the message space with the same rank. Given an OPE-encrypted column, in cumulative attack, the adversary is able learn both the frequencies and the relative ordering of the encrypted values. Combining orders with frequencies, the adversary can easily tell for each ciphertext c what fraction of the encrypted values are less than c.

Leakage-Abuse attacks Cash et al. [20] study the leakage of SE schemes. They first present a characterization of the leakage profiles of SE schemes. They divide the leakages in four categorizes– $L4$, $L3$, $L2$ and $L1$. In $L4$ leakage, the adversary has full plaintext information under deterministic word-substitution cipher. It learns two kind of information. The first is the pattern of locations in the text where each word occurs and the second is its total number of occurrences. Moreover, it gets frequencies of the unique indexed keywords, lengths of the ciphertexts, and the order of appeared keyword, immediately upon upload. Thus it reveals a co-occurrence pattern and is vulnerable to frequency analysis.

In $L3$ leakage, the adversary fully gets occurrence pattern with keyword order which reveals the pattern of keyword occurrences in the documents, in the order of their first appearance, but not the *occurrence counts within a document*. In $L2$ leakage, the adversary fully gets an occurrence pattern. This is similar to $L3$, however, in $L2$ document order is not revealed. The $L1$ leakage is the strongest one. It is similar leakage as $L1$, but only for terms that have been queried.

For the SE schemes having $L3$ and $L2$ leakage pattern, they finally show plain-text can be recovered partially when the adversary has knowledge of a set of documents. From the partial knowledge of the database, they show queried keywords can be recovered from the SE schemes having $L1$ leakage.

File-injection attacks Zhang et al. [110] studied file-injection attacks that focus to recover keywords from future queries. The idea was to construct a set of files with certain properties depending on attack strategies, inject them into the server database add query and learn from the leakages.

However, if the initial set of keywords is large, then the attacker needs to inject an exponential number of large files which makes the attacks weak.

In the first type of attack, i.e., binary search attack, it inserts $\log K$ number of files where $K = |\mathcal{W}|$. The ith file contains exactly the keywords having ith most significant bit as 1. If a keyword w is searched and returns then it matches returned files with its injected ones. The number of required files to be inserted can be reduced by considering a targeted keyword set smaller than $\mathcal{W}$. Later they proposed several advanced attacks. In a hierarchical-search attack, the adversary makes a partition of $\mathcal{W}$ and executes a binary search attack for each partition. This significantly reduces the number of files injected.

In the above attacks, the adversary does not have prior knowledge of its plain text. When the adversary has partial knowledge of plain-text information such as the co-occurrence matrix, the number of required injected files is reduced further with significant accuracy. Later they suggest padding each file with extra keywords that protect a SE scheme from such attacks. The attacks are applicable to conjunctive search SE schemes too.

The file injection attack considers that the adversary is able to force the owner to inject its selected set of documents. However, *the attacks can be prevented when the scheme is forward private* where a newly added document cannot be related to any past search result until the next search occurs. The attack forces SE schemes to be forward private where file injection is applicable.

Passive leakage-abuse attacks In 2017, Giraud et al. [37] refine leakage profiles $L4$, $L3$, $L2$ and $L1$ given in [20]. Then they present three partial plaintext recovery attacks on $L4$, $L3$, and $L2$ leakage profiles which are claimed to be more practical than [20]. The attacks assume the knowledge of some amount of sample plaintexts. We briefly describe them as follows.

The first attack is *Mask attack* on $L4$-SSE schemes. Here, the adversary can find encryption of a file by just looking at the appearance of the keywords in an existing encrypted document. From this, it can match plaintext and their encrypted versions. In the second attack, i.e., *Co-mask attack* on $L4$-SSE schemes, the adversary matches co-occurrences of the known plaintext documents with the encrypted documents.

In the third attack, i.e., *PowerSet attack* on $L2$-SSE schemes, where Co-mask attack can no be performed as the order of the keywords are not preserved here, it works in two different ways– associates documents to its identifiers and associates keywords to its tokens.

3.1.4 Forward and backward secure search schemes

A forward private DSE scheme does not leak any information about the previous search results when new documents are added. We have seen that a dynamic search scheme should be forward private in order to be protected from file-injection attacks. The concept of forward privacy is introduced by Stefanov et al. [87]. Previous to their scheme, Chang and Mitzenmacher [26] gave a forward secure scheme which was published way ago. Though Stefanov et al. [87] introduced a forward secure scheme, their proposed scheme was ORAM-based.

ORAM-based schemes Goldreich and Ostrovsky [41] first introduced Oblivious RAM (ORAM). Naveed et al. [71] developed a methodology to study how ORAM can be used is DSSE schemes, reducing access pattern leakage. They showed that it is not possible to completely eliminate leakage in an SSE scheme without downloading the entire outsourced data. Besides ORAM based search scheme proposed by Rizomiliotis and Gritzalis [79], Garg et al. [36] proposed a TWORAM and Path-ORAM based SSE scheme they proposed a scheme that does not leak the search pattern. In 2004, Boneh et al. [11] presented a searchable encryption scheme using public-key encryption techniques for static data. Due to the higher complexity of the public-key encryption scheme, it has high computational costs.

Trapdoor permutation based forward private keyword schemes Bost [16] proposed a scheme $\sum$oφoς that was not ORAM based. The scheme is designed with the help of trapdoor permutation.

He proposes two schemes, a basic version $\sum$oφoς-B that supports addition only and $\sum$oφoς that supports deletion too with extra cost. $\sum$oφoς-B uses two tables W (kept in client) and T (outsourced to cloud server). W keeps a counter c and the state information ST_c corresponding to each keyword. Initially $c = 0$ and ST_0 are chosen at random by the client. A keyword-document pair (w, id) is added in T as $T[UT_i] \leftarrow id \oplus H_2(K_w, ST_i)$ where $UT_i \leftarrow H_1(K_w, ST_i)$ and $ST_{i+1} = \Pi_{sk}(ST_i)$, where Π_{sk} is a random permutation. For searching, the client sends K_w, ST_c, c.

For deleting a document, the client needs to know the keywords of the file to be deleted. So, before deletion, the deleted file should be downloaded and decrypted by the client. This adds an extra round of communication. Moreover, they proposed another version $\sum$oφoς-ϵ in which a tag corresponding to the each keyword w is stored makes the scheme verifiable. However, it is not clear how to store compute tags in $\sum$oφoς-ϵ, and has no formal proof of verifiability. The scheme also does not have any description of how the verification will work in the case of a dynamic database. It can use multiset hashing for the same which makes the scheme dynamic. In that case,

the owner doesn't have to search for the existing file identifiers while adding a keyword. Moreover, the client needs to store the tag of the set of identifiers corresponding to each keyword.

Backward privacy A backward privacy (see Section 1.1.7) do not reveal the information about the deleted identifiers.

So far, though the forward privacy has been explored by many researchers, not so is with the backward privacy. This might be because there is no formal attack for a scheme that is forward private but not backward private.

Forward and backward privacy together Bost et al. [18] explore backward privacy extensively. They define three kinds of backward privacy- Type-I, Type-II, and Type-III. A Type-I backward private scheme leaks the documents identifiers that currently contains the searched keyword, the insertion time, and the count of updates on the keyword. A Type-II backward private scheme leaks when all the updates on w happened in addition. However, the contents of the updates are hidden here. A Type-III backward private scheme leaks which deletion update canceled which insertion update in addition to the leakage by a Type-II backward private scheme. Thus Type-I is the strongest and Type-III is the weakest notion of privacy among them.

Bost et al. [18] present several schemes that achieves both forward privacy and different types of backward privacy. They first describe a generic backward private scheme $B'(\Sigma)$, where Σ is an arbitrary SSE scheme, with the cost of one extra round of communication during the search. The main idea of the scheme is that when a keyword is searched, the returned keyword-document pairs are re-encrypted with a new version of the key corresponding to the searched keyword, and then the new encrypted pairs are uploaded. Thus it does not keep previous search results and hence no leakage with updates. They instantiate Σ with different schemes. For example, when Σ is a result-hiding scheme B' achieves Type-II backward privacy. Its TWORAM [36] instantiation, Moneta, achieves Type-I backward privacy, whereas its $\sum o\varphi o\varsigma$ [16] instantiation, Fides, achieves that of Type-II.

Then Bost et al. [18] define the Forward Secure-Range Constrained PRF (FS-RCPRF) framework that builds a single-keyword forward-private SSE scheme from any range constrained PRFs (CPRF). Diana, the GGM [40] instantiation of CPRF, is a forward-private scheme whereas its modified versions Diana$_{del}$, that supports deletion, is a two-roundtrips Type-III backward-private scheme. Finally using the puncturable encryption schemes, they design another framework Janus, with incremental update property, which is forward-private as well as achieves Type-III backward privacy.

Forward and Type-I$^-$ backward privacy Zuo et al. [114] later introduce Type-I$^-$ backward privacy that is stronger than Type-I backward privacy. Given a time interval between two search queries for a keyword w, a Type-I$^-$ backward private SSE scheme leaks the files that currently match w and the total number of updates for w. Unlike Type-I schemes, it does not leak when the files are inserted. Later they present a forward and backward private DSSE (FB-DSSE) scheme which is forward-private as well as Type-I$^-$ backward private. The scheme is based on $\sum o\varphi o\varsigma$ [16] and is designed by leveraging simple symmetric encryption with homomorphic addition and bitmap index.

Their proposed scheme only returns identifiers of the result files. However, in searchable encryption schemes, retrieval of the result files and leakage due to retrieve are considered. This separates their scheme from other DSSE schemes.

Forward privacy in conjunctive queries Wu and Li [99] propose an efficient conjunctive search scheme that achieves forward privacy, called virtual binary tree or VBTree. The tree is inspired from KRB Tree [47] and IBTree [56]. As in the KRB tree, the files identifiers are mapped with integers from zero in ascending order. Then a binary tree is prepared where file identifiers are kept in the leaves. The set of keywords belonging to a file are kept in all internal nodes in the path of the file. Note that, in the VBTree construction, an internal node does not need to store the same keyword if more than one file below the node contains it. Also, this reduces the cost of storage compare to the KRB tree.

The mechanism to keep keywords in internal nodes is inspired from [56]. When some file is added, a keyword belonging to the file is added only in the internal nodes in the path from the root to the leaf where the keyword is not present before. So, when a new file is added, the addition process does not leak information until the next search. This makes the scheme unique and forward private.

Secondly, the VBTree is stored only conceptually as a tree, but the tree structure is completely hidden. The cloud sees only a set of key-value pairs, nothing more than that. However, when some search is performed, some controlled part of the tree is revealed to the cloud. Moreover, the use of only symmetric key encryption schemes makes the scheme efficient and practical.

Other forward and backward private schemes Bost et al. [18] have formalized definitions for three forms of backward privacy and provided a solution for each of them using constrained PRFs. Thereafter, Chamani et al. [25] have formalized leakage function for those forms. They have proposed a non-interactive backward-secure search scheme from symmetric puncturable encryption

which is an improvement over [18]. At the same time, Sun et al. [90] have introduced symmetric puncturable encryption and using it proposed another improved backward secure scheme that can revoke a cloud server's searching ability on deleted data.

3.1.5 Verifiable Searchable Encryption schemes

In most of the works, the cloud service providers are considered semi-honest i.e., honest to follow the protocol but curious about the data and queries. That is why they become vulnerable when the cloud server behaves maliciously. Being verifiable, an SE or a DSE scheme becomes protected from such cloud servers. Here we describe some of the works on verifiability.

Verifiable query over static data Chai and Gong [24] first introduce verifiable SE schemes. The scheme is for single keyword search and supports only static data. For verification, it uses a trie-like dictionary data structure where the search is performed in logarithm time in terms of length of the keywords. In the tree, each leaf node indicates some keyword. In the verifiable SE scheme, a leaf node stores all the identifiers that contain the corresponding indicating keyword. A keyed hash function is used that binds the file identifiers with the parent node in the path to the keyword. The cloud returns the path to the client with the search results. Since the client stores the key, it can easily recompute the hashes and can verify if any data changed in the path.

Later Wang et al. [96] propose a verifiable search scheme that works for conjunctive search. The scheme is like Cash et al. [21] scheme. An accumulator is generated for each keyword which helps to prove whether a returned result really exists. An authentication tag is also generated that ensures that the list of encrypted identifiers corresponding to a keyword is unchanged. Finally, an accumulator is generated for a set of size $|\mathcal{W}|$, and another accumulator of the size of the number of keyword-document pairs is generated.

However, the scheme is only for static data. It assumes the keywords to be unique primes. Generating this large number of primes is a hard task. It has referred Sun et al. [89] to find such primes. It considers the least frequent keywords while answering a query. However, *it has not mentioned how keyword frequency record is kept and retrieved during a query*. So, as in [21], it either requires extra storage at the client-side or requires an extra round of communication before every query.

In most of the SSE schemes, the set of keywords i.e., the dictionary is fixed. In case, while searching a keyword out of the dictionary, the cloud may not return the correct proof of non-existence. Ogata and Kurosawa [74] propose a single keyword search generic verifiable SE scheme

that returns correct proof for any searched keyword string. The scheme supports only static databases and uses a cuckoo hash table in which search time is constant. It uses two tables and two hash functions for them but keeps the value in only one table that is determined by the cuckoo hashing algorithm.

The main idea of the scheme is that, for each keyword w, a keyed hash of the keyword w is computed. Then another keyed hash of $DB(w)$ bounded with the hash of the keyword is computed. During the search, Both values are returned from their stored values in the cuckoo hash table. If the keyword is present, the cloud server returns correct values from the cuckoo hash table and the client can verify. If the keyword is not present. Then there will be a failure to match keyed hashes.

Multi-set hashing and Merkle tree based verification The scheme by Chai and Gong [24] is not forward private. Bost et al. [17] first study the searchable encryption schemes from the theoretical point of view. They give lower bounds on the computational complexity of search and update queries. The bounds are intuitive for semi-honest adversaries. In presence of such adversaries, they show the search cannot be done in less than $\Omega(m)$ where m is the number of results for a query, and the update has to run in $\Omega(1)$ per modified document/keyword pair computationally. From a storage point of view, they show that if the client's private storage is not linear in $|\mathcal{W}|$ in the database, the verification of an SSE scheme has to have a logarithmic overhead for either search or update queries. They present two generic solutions that match these lower bounds.

The first solution is based on multi-set hashing and Merkle tree-like data structures. They present the idea of the verifiable hash table (VHT) which has performance like a normal hash table, however, it additionally returns proof that the search returns the correct result. The key idea of a VHT is that it sorts all key-value pairs, then computes the MAC of each pair together with rank in the sorted list. During the search, the cloud returns the pairs before and after of the searched one together with rank. From this information, the client can verify easily. The generic construction GSV of verifiable SE uses a multiset hash for each keyword. The hashes are then kept in a VHT. They also compared the performance of a GSV with accumulator-based VHT. Though the scheme achieves good performance, it is not verifiable. Finally, using VHT they extend Stefanov et al. [87] to be verifiable keeping forward privacy property unchanged.

Verifiable conjunctive keyword search According to Wang et al. [96], they are the first to design a Static Verifiable SSE scheme that supports conjunctive keyword search and *can handle large database*. The scheme is an extension of Cash et al. [22] and Sun et al. [89] schemes. The verification process of the scheme is based on Bilinear-map Accumulator.

The idea of the scheme is as follows. For each keyword, a list of identifiers of documents are encrypted individually and kept in a table $TSet$ where all such encrypted document identifiers are kept. Then for the list, an accumulator value is generated. The accumulator value together with its authentication tag is kept in the table $TSet$. Besides, corresponding to either a keyword-document pair, an $xtag$-value is computed and kept in a table $XSet$. Similarly, corresponding to every keyword, a tag value $stag$ is computed and kept in a table $Stag$. Finally, an accumulator is computed for each of the tables.

During the search, given a set of keywords $\{w_1, w_2, \ldots, w_n\}$, the cloud at first finds the list of identifiers R_{w_1} containing w_1 and checks its authenticity. Then for each identifier in R_{w_1} checks whether all other searched keywords are present or not. For the same, it regenerates $xtag$-values for the keyword-document pairs. From the accumulator values, the cloud returns proof of either existence or non-existence of the pairs.

The search cost of their scheme depends on the number of documents matching with the least frequent keyword. The scheme can achieve verifiability of the search result with constant size communication overhead between the client and cloud server. The security of their scheme is based on q-SDH assumption (see Definition 2.15). However, the scheme works only for static databases. They assume that each keyword should be a prime, which led to $O|\mathcal{W}|$ extra cost of keeping the list of the primes together keywords at the client-side.

UC-secure verifiable conjunctive search Sun et al. [91] present a verifiable conjunctive keyword search (VKCS) scheme which is based on bilinear map accumulator and accumulation tree. The basic idea of the scheme, to generate the secure index, is as follows. It considers an $m \times n$ matrix where m is the number of unique keywords and n is the files. Each row and column corresponds to a keyword and a file respectively. Thus an entry is 1 if the keyword-file pair exists, otherwise zero. Each entry is then encrypted with some pseudo-random bit. Then for each keyword, an accumulator is generated for the entries with value 1 i.e., for the files containing the keywords. Finally, an accumulation tree is prepared for the accumulators. During the search, the rows of the searched keywords are recovered, decrypted, and then computed intersection. From the stored accumulator and accumulation tree, the verification can be done publicly. To add, delete or modify a file, the client does the same for the corresponding column.

Later, they show the scheme is UC-secure against a malicious adversary. Though the scheme supports dynamic updates, it is not forward private. The scheme is easier to implement. For each file, it takes constant space of $|\mathcal{W}|$-bits in the secure index. So, if the dictionary size is large, and the files are small, then the matrix become sparse resulting good amount of storage expansion.

Verifiable search with forward privacy Using algebraic PRF, Yoneyama, and Kimura [107] propose a verifiable SSE scheme that is dynamic as well as forward private. The APRF is a special type of PRF such that certain algebraic operations on these outputs can be computed more efficiently with the secret salt than computing separately. The general idea of verifiable schemes is to make a tag of the identifiers, corresponding to each keyword, containing that keyword. However, tag generation algorithms are different. [107] uses APRF for tag generation on top of [16] keeping the forward private property unchanged.

For each keyword w it generates verification tag $Ver_{c+1} \leftarrow AF(K_V, (c_{(w)}+1, w, ind))$ for the file ind containing it, where K_V is the verification key and c_w is used as counter. The client sends them to the cloud together with keyword-file pairs of the base scheme [16]. During search, the cloud collects all such tags for the searched keyword and sends $Ver^w = \prod_{i=0}^{c} Ver_i$ to the client together with set of identifiers $DB(w) = \{ind_i,\ 0 \leq i \leq c\}$. From the result identifiers and the aggregated product Ver^w of verification tags, the client can efficiently check the validity of the result as it stores the counter as the state of the database. Since the client does not store any tags at its own place, the storage cost is reduced here from the earlier single keyword search verifiable scheme. However, in the scheme, server-side storage is increased while achieving the above.

bilinear map accumulator and Merkle hash based conjunctive search verification Li et al. [57] propose a scheme conjunctive keyword search scheme. It uses a bilinear map accumulator and a Merkle hash tree for verification. The basic idea of the scheme is inspired from [48] where two types of indices are used for searching- inverted index and general index. However, the scheme is only for a single keyword search. Li et al. [57] generalize the scheme that supports conjunctive search with verifiability.

During building, first, a search index is generated similarly as [48]. Then for every keyword, an accumulator is generated for the set of file identifiers containing the keyword. A Merkle hash tree is generated that ensures the integrity of such accumulators.

On a search request, the cloud performs the single keyword query search for all keywords in the query. Then the intersection of the result is computed. Finally, the cloud server gives the proof of two things; subset and completeness. The result set i.e., intersection set, is the subset of all single keyword search results. The cloud gives proof of membership of the intersection set in all resulted sets. This is possible as for every keyword, an accumulator is saved for all documents containing the keyword. The proof of completeness is given using the extended Euclidean algorithm for polynomials.

Despite the scheme uses the symmetric key encryption scheme for search and supports dynamic

updates, the use of the public key encryption for search result verifiability makes the scheme heavier. Another drawback of the scheme is that it has to perform a single keyword search for all keywords in a conjunctive query.

Publicly verifiable Boolean query Jiang et al. [45] present a good Boolean search scheme that is publicly verifiable too. The base of the scheme is based on OXT protocol [22]. For verification, they use accumulators together with a special Merkle hash tree where the information for verification is stored. The Merkle hash tree converts a Boolean keyword search operation in a DSE to a set operation.

As OXT protocol, for each keyword-document pair, a unique key-value pair is created and stored in a table. Then the search and update protocol for searching is similar as [22]. For verifiability, each pair is mapped to an element in a bilinear group. Then, an accumulator is generated for such keywords belonging to the same document. Such accumulators become elements of the next-level accumulators. Thus, any non-leaf node is an accumulator of its children, whereas a leaf node corresponds to some keyword-document pair. During verification of a search result, with the help of the root of the tree, that is the digest of the tree, the client can verify easily.

In this scheme, an update requires two rounds, which is a drawback of the scheme. However, any accumulator-based scheme requires two rounds of communications if the owner does not store the state information of the database.

Other verifiable search schemes Cheng et al. [28] have presented a VSSE scheme for static data based on the secure indistinguishability obfuscation. Their scheme also supports Boolean queries and provides publicly verifiability on the return result. With a multi-owner setting, Liu et al. [63] have presented a VSSE with aggregate keys. Miao et al. [68] presented a VSSE in the same multi-owner setting. All of those schemes are for static databases and are privately verifiable where the VSSE schemes by Soleimanian and Khazaei [85] and Zhang et al. [109] are publicly verifiable.

There are some works that deal with complex queries when the data is static. Conjunctive query on static data has been studied by Miao et al. [69], Wang et al. [96], Miao et al. [67] etc. These schemes have private verifiability. [102] is a blockchain-based scheme that supports Boolean range queries keeping the encrypted data index and queries on a blockchain having a good amount of monetary cost for each search whereas [6] only supports conjunctive search.

Dynamic verifiable SSE with complex queries also has been studied. Zhu et al. [112] presented a dynamic fuzzy keyword search scheme that is privately verifiable and Jiang et al. [44] has studied Publicly Verifiable Boolean Query on dynamic databases. For dynamic data, a dynamic fuzzy

keyword search scheme was proposed by Zhu et al. [112] which is privately verifiable. Again, publicly verifiable dynamic SE scheme by Jiang et al. [45] allows the query to be Boolean where Sun et al. [91] allows only conjunctive searches.

3.2 Queryable Encryption on graph data

Graph algorithms are well studied when the graph is not encrypted. Since the necessity of outsourcing graph data in encrypted form is increasing very fast and encryption makes it difficult to work those algorithms, a study is required to enable them. There are only a few works that deal with the 'query' on 'outsourced encrypted graph'.

Encrypting static graph Chase and Kamara [27] discuss graph encryption while introducing the notion of structured encryption as a generalization of searchable symmetric encryption (SSE) proposed by Song *et al.* [86]. They present queryable encryption schemes for the queries that include lookup queries on matrices, search queries on labeled data, neighbor queries and adjacency queries on graphs, focused subgraph queries on labeled graphs.

Given a pair (α, β) lookup queries on matrices M returns the value of $M[\alpha, \beta]$. The encryption of such matrix consists of permutations of rows and columns and encrypting individual entries. For example, the entry $i = M[\alpha, \beta]$ is kept at $M'[\alpha', \beta'] = \pi(i) \oplus F_{k_1}(\alpha, \beta)$ where, F is a PRF, M' is the new encrypted matrix, (α', β') is a pseudo-random permutation of (α, β) and $\pi(i)$ is an encrypted value of i using a symmetric key encryption. In case the graph is labeled i.e., $(i, v_i) = M[\alpha, \beta]$, the new entry will be $M'[\alpha', \beta'] = (\pi(i), v_i) \oplus F_{k_1}(\alpha, \beta)$.

In the case of a neighbor query, instead of storing the encrypted values in M', all of them corresponding to a vertex are stored as a tuple. The tuple is stored in a dictionary where the key of the key-value pair is generated from the vertex. For a directed graph, given a vertex, a focused subgraph query returns all the vertices that are connected either to or from the given vertex. For a focused subgraph query, they keep two encrypted graphs- one only for the to vertices and the other for the from vertices. On query, the cloud server returns combining both the results.

In all of their proposed schemes, the graph is considered as an adjacency matrix and each entry is encrypted separately using symmetric key encryption. So, the scheme is inefficient for a large sparse graph as the storage requirement grows faster. Moreover, the scheme is only for static graphs.

Privacy preserving queries Different types of privacy-preserving queries on encrypted graphs are studied by Cao et al. [19], Zhang et al. [108]. SPARQL query on encrypted graphs is studied by Kasten et al. [49]. Based on [32], Xu et al. [104] proposed a scheme to find all paths, from a source to a destination vertex, in a privacy-preserving manner. Lai et al. [54] proposed a graph encryption scheme with the ability of social search which allows users to search the content of interests created by their friends. Xie and Xing [101] have studied clustering coefficient. They encrypt each entry of the adjacency matrix, using homomorphic encryption, making the scheme inefficient for large datasets and frequent queries.

Besides, Zheng *et al.* [111] proposed link prediction in decentralized social networks preserving privacy. Their construction split the link score into private and public parts and applied sparse logistic regression to find links based on the content of the users. However, the graph data was not considered to be encrypted in the privacy-preserving link prediction schemes.

Approximate shortest distance queries Sketch-based oracles are useful to find an approximate shortest distance between two vertices, with some error bound. Meng *et al.* [65] study the same for encrypted outsourced graphs. They consider static graphs only. They compute most of the steps at the pre-processing stage where the client computes the sketches for every vertex that is useful for efficient shortest distance query. Instead of encrypting the graph directly, they encrypted the pre-processed data.

They use two sketch-based oracles. However, the concept is the same for both. For each vertex v, a set of vertex-distance pairs $\{(w_z, \delta_z) : z = 1, \ldots, \lambda\}$ is found, where w_zs are representative nodes with δ_zs being the distance from v. During a search, given two vertices, a common node is found between the sketches of them. The sum of distances of the common vertices is returned as result. To encrypt the distances in the vertex-distance pairs in a sketch, they use a somewhat homomorphic scheme that enables adding the values without decrypting them. The client can decrypt the approximate shortest distance from the encrypted result.

However, the encryption scheme only encrypts the required sketches and does not preserve the graph structure. Thus, in their scheme, there is no chance of getting information about the original graph. The basic queries like vertex queries or edge queries cannot be performed here.

Exact shortest distance queries The previous work [65] finds an approximate shortest distance between two points. It does not ensure if we found two common nodes in their sketches and then no result can be found. Wang et al. [97] try to solve the problem and find the exact distance between two points in encrypted weighted graphs. Moreover, the scheme is dynamic.

Their scheme is inspired from [48] that allows it to be dynamic. They use Dijkstra's algorithm to find the shortest distance. If we see [48], given a search query token, the cloud can traverse only to its adjacent vertices. However, Dijkstra's algorithm requires the cloud to traverse any depth until it reaches the destination vertex. So, the search token for all the adjacent vertices is stored with the adjacent vertex set information. Thus, given a vertex, the cloud can return any vertex connected to it.

The cloud has to compare the weight of the edges that are encrypted. To compare them, the client uses another proxy server and gives it a partial key to decrypt the weights. The cloud and the proxy server compares the weights using a garbled circuit (see Section 2.18) without knowing them.

However, one of the major disadvantages is that in a single query, the scheme leaks all the nodes reachable from the queried vertex which is a lot of information about the graph. For example, if the graph is complete, it reveals the whole graph. Though the scheme uses private key encryption for all, it requires the help of an extra proxy server that computes the garbled circuit and makes the scheme heavy for the cloud.

Forward private search over encrypted labeled bipartite graphs Lai and Chow [34] construct two forward private DSE schemes over labeled bipartite graphs. The schemes are a generalization of single keyword search SE schemes. The first is a generic construction from any DSE, and the other is a concrete construction from scratch. Besides neighbor queries, our schemes support flexible edge additions and intelligent node deletions. The cloud server can delete all edges connected to a given node, without having the client specify all the edges. The scheme makes any DSE scheme forward private and inspired from [42]. And we see that the problem of single keyword search is similar to find adjacent vertices in a bipartite graph where two distinct sets are the set of keywords and the set of documents. The main idea is to locally maintain a table γ of PRF keys K_x and counters c_x for each query $q = x$ so that adding an edge (x, y, w) updates the counter and the key used for the query is a function of both K_x and the counter. So, there is no repeatation for the same keyword.

They design a novel data structure called cascaded triangles in which traversals can be performed in parallel while updates only affect the local regions around the updated nodes. They present the second scheme using cascaded triangles. The main idea is that instead of keeping the set of neighbor in a single linked list, they are put it in a complete binary tree. This enables the search, in a different branch, to be performed in parallel.

Approximate constrained shortest distance queries Shen *et al.* [84] propose an approximate *constrained shortest distance* (CSD) query scheme $Connor$ in encrypted graphs which finds the shortest distance with a constraint such that the total cost does not exceed a given threshold.

They use a somewhat homomorphic encryption scheme and order revealing encryption scheme. They design a tree-based ciphertexts comparison protocol, which helps to determine the relationship of the sum of two integers and another integer over their ciphertexts with controlled disclosure.

The main difference between [84] and [65] is that, in [84], for each node in and out vertices are stored in different structures. Given a node, the cloud can traverse in both directions whereas it can go in out directions only in [65]. Sketches are used in pre-processing stage for both of them. Secondly, instead of Dijkstra's algorithm, it [84] traverse in a tree structure called *cost constraint tree*. For comparison, they used order revealing encryption that helps to compare the distances and find the approximated shortest one.

Parallelizable graph encryption A parallel secure computation framework *GraphSC* has been designed and implemented by Nayak *et al.* [73]. This framework computes functions like histogram, PageRank, matrix factorization, etc. To run these algorithms, *GraphSC* introduced parallel programming paradigms to secure computation. The parallel and secure execution enable the algorithms to perform even for large datasets. However, they adopt Path-ORAM [88] based techniques which is inefficient if the client has little computation power or the client doesn't use very large size RAM.

Chapter 4

Efficient Keyword Search on Encrypted Dynamic Cloud data

One of the primary objectives of designing an SSE scheme for cloud data is to reveal as little information to the cloud server as possible. A naive solution is the following. For every search query, the client downloads the complete encrypted data, decrypts them, and performs the search on the data. However, downloading the whole data for each search query is impractical and defeats the purpose of outsourcing data to the cloud server. SSE schemes on encrypted data handle keyword searches more efficiently than this naive solution. On the other hand, uploading data to the cloud in an encrypted form is not enough. In the case of dynamic data, the client may need to add or delete data. In such a scenario, the SSE scheme should be able to support database updates. An SSE scheme that supports such updates is said to be a Dynamic SSE (DSSE) scheme.

A DSSE scheme can have the access pattern leakage, search pattern leakage, etc. (see Section 1.1.6). For any DSSE scheme, it is desirable to reduce these leakages. Although some DSSE schemes [79, 36] based on oblivious RAM [41] provide maximum privacy (i.e., minimum leakage), they require huge bandwidth and storage — which makes these schemes inefficient. Some other DSSE schemes, like [48, 42, 21], compromise privacy at the cost of efficiency.

With search pattern leakage, Zhang et al. [110] showed a file-injection attack (see Section 3.1.3) can recover queried keywords with high efficiency. In such an attack, the adversaries can force the clients to add any number of files of its choice. Forward private schemes, where previously searched keywords are not revealed during adding a new file, are protected from such attacks. So far, there is no practical attack over the schemes which have only forward privacy but not backward privacy (3.1.4). In a backward private DSSE scheme, within any period, if two search queries on the same keyword happened, it does not leak information about the files that have been previously added and later deleted [18].

However, the file-injection attacks [110] rely on the assumption that the adversary (cloud) can inject files in the encrypted database using the owner's key which is the same as giving the key to the cloud or stealing by the adversary. The first case can be avoided simply by not considering such email-like databases where key is given to the adversary. In the second case, it is difficult to protect if not any encryption scheme if the secret key is stolen by the adversary. Hence making a

scheme forward or backward private is not always necessary by increasing the other costs.

Our contribution We summarize our contribution in this chapter as follows.

- We present a DSSE scheme Trids for cloud data that
  - has a keyword search time linear in the number of documents containing the searched keyword.
  - has a file update time linear in the number of keywords present in the file — which is optimal in the sense that the list of files corresponding to each of these keywords must be accessed and updated accordingly.
  - requires less storage overhead than the existing schemes like [48].
  - does not leak, in case of file deletion, the keywords that have not been searched previously.

  A detailed comparison with existing DSSE schemes (based on these parameters) for cloud is given in Section 4.3.
- Our design exploits efficient data structures that significantly reduce the storage, search time, and update time. We provide a simulation-based security proof of our DSSE scheme Trids.
- We build a prototype of our scheme and run experiments on real-life data sets [33]. For the results, we observe that even with a dictionary having 450,000 keywords and 2,300,000 keyword-file pairs, search time is only around 300 μs — which shows the efficiency of our DSSE scheme Trids.

We also compare the performance of our scheme Trids with those of other DSSE schemes found in the literature (see Section 4.3). For a quick review, we refer to Table 4.2 and Table 4.3 that provide a comparison between Trids and other DSSE schemes based on different parameters. From these tables, we observe the following.

- The scheme presented in [42] has a smaller index size compared to our scheme Trids. However, in Trids the client stores no data (except the private key), whereas [42] has to store a search history of size $O(N)$ at the client's end.
- Although the DSSE scheme presented in [87] has less leakage, it has $O(\log N)$ communication rounds with high bandwidth and large client-side storage which makes the scheme less desirable.

- Although [16] has less index size compared to us, it has large client-side storage even with a good amount of cloud side computational cost.
- We note that other DSSE schemes [18, 90, 25] have been proposed recently which are forward and backward private and protect the schemes from file-injection attack [110]. However, the requirement of large client storage (at least $O(|W|)$) makes the schemes inappropriate for lightweight clients.
- Our scheme Trids has significantly less storage and less leakage than the "state-of-the-art" DSSE scheme [48].

Organization The rest of the chapter is organized as follows. We give some preliminary definitions in Section 4.1. We describe our proposed scheme in Section 4.2. In Section 4.3, we discuss a comparative study of our scheme with existing similar schemes. The security of our proposed scheme is described in Section 4.4. We evaluate the performance of our scheme in Section 4.5.

4.1 Preliminaries

Let $\mathbf{f} = \{f_1, f_2, \ldots, f_n\}$ be a collection of n documents or files. c_{f_i} is the encrypted version of f_i and $\mathbf{c} = c_{f_1}, c_{f_2}, \ldots, c_{f_{|\mathbf{f}|}}$ is the collection of encrypted files.

For each file f, there is a file identifier $id'(f)$. Instead of the original name, a keyed hash function id' is used where the key is secret to the client. This is because sometimes the name of a file leaks information about the content of the file. Similarly, instead of dealing with the keywords directly, a keyed hash function id is used and identifiers of them are generated. The set of files containing a keyword w is denoted by f_w and I_w refers to the set of file identifiers containing a keyword w. Table 4.1 contains the notations used in the chapter.

4.1.1 DSSE Scheme

Initially, in a DSSE scheme, the client takes a set of files $\mathbf{f}$ and generates an encrypted inverted index γ and a set of encrypted files $\mathbf{c}$. *Inverted index* is a type of data structure where for each keyword, the set of identifiers of the documents, that contains the keyword, is kept. Then the client uploads $(\gamma, \mathbf{c})$ to the cloud server. Later, the client generates tokens for adding (or deleting) files or searching for keywords. A *token* is an encrypted trapdoor that is generated using the client's secret key and a query (either search or add or delete query). It helps the cloud server to perform

Table 4.1: Notations used for proposed keyword search scheme

Symbols	Meaning
F	PRP $\{0,1\}^{\lambda} \times \{0,1\}^{\log \lvert W \rvert} \rightarrow \{0,1\}^{\log \lvert W \rvert}$
F'	PRP $\{0,1\}^{\lambda} \times \{0,1\}^{\log l_{max}} \rightarrow \{0,1\}^{\log l_{max}}$
G	PRG $\{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\log \lvert W \rvert}$
G'	PRG $\{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\log l_{max}}$
N	A cell of the array A
L, R, D	Pointer to the left/ right/ down node in TDL
W	Set of all possible keywords/ Dictionary
$id(w)$, $id'(f)$	Identifier corresponding to the keyword w / file f
$P(,)$	A PRG $\{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\lambda}$
t_a, t_s, t_d	Add, search and delete token respectively
λ	The Security parameter
$\hat{f}$	Set of distinct keywords in the file f
H_i	Standard hash function like SHA-256 ($i = 1(1)5$)
k_w, k_f	Key corresponding to the keyword w/ file f
l_{max}	Maximum #files that can be stored by the cloud

the query on the encrypted index γ without knowing the query. Finally, the cloud server sends the query results to the client and updates $(\gamma, \mathbf{c})$ accordingly. We define an index-based DSSE scheme in Definition 4.1 which follows from [48].

Definition 4.1 (DSSE scheme). *An index-based Dynamic Symmetric Searchable Encryption Scheme Σ is a tuple Σ =($KeyGen$, Enc, $Build$, $AddToken$, Add, $SearchToken$, $Search$, $DeleteToken$, $Delete$, Dec) of algorithms defined as follows.*

- $K \leftarrow KeyGen(1^{\lambda})$*: is a client-side PPT (probabilistic polynomial-time) algorithm that takes λ as security parameter and outputs a secret key K.*
- $c_f \leftarrow Enc(K, f)$*: is a client-side PPT algorithm that takes a secret key K and a file f as input and outputs the encrypted file c_f of f.*
- $(\boldsymbol{c}, \gamma) \leftarrow Build(K, \boldsymbol{f})$*: is a client-side PPT algorithm that takes a key K and a set of files $\boldsymbol{f}$ as input and outputs the encrypted set of files $\boldsymbol{c}$ and an encrypted inverted index γ.*
- $t_a \leftarrow AddToken(K, f)$*: is a client-side PPT algorithm that takes a secret key K and a file f as input and outputs an add token t_a.*
- $(\boldsymbol{c}', \gamma') \leftarrow Add(t_a, c, \boldsymbol{c}, \gamma)$*: is a server-side PPT algorithm that takes an add token t_a, an encrypted file c, a set of encrypted files $\boldsymbol{c}$ and an inverted index γ as input and outputs*

updated $\mathbf{c}'$ *and* γ'.

- $t_s \leftarrow SearchToken(K, w)$*: is a client-side PPT algorithm that takes the secret key* K *and a keyword* w *as input and outputs a search token* t_s.
- $I_w \leftarrow Search(t_s, \gamma)$*: is a server-side PPT algorithm that takes a search token* t_s *and the index* γ *as input and outputs* I_w*. It contains the identifiers of the files that contain* w.
- $t_d \leftarrow DeleteToken(K, id'(f))$*: is a client-side PPT algorithm that takes the key* K *and a file identifier* $id'(f)$ *and outputs a delete token* t_d.
- $(\mathbf{c}', \gamma') \leftarrow Delete(t_d, \mathbf{c}, \gamma)$*: is a server-side PPT algorithm that takes a delete token* t_d*, a set of encrypted files* $\mathbf{c}$ *and an inverted index* γ *as input and outputs updated* $\mathbf{c}'$ *and* γ' *after deletion.*
- $f \leftarrow Dec(K, c)$*: is a client-side PPT algorithm that takes the secret key* K *and an encrypted file* c *as input and outputs the decrypted file* f.

Correctness: A DSSE scheme Σ is said to be $correct$ if $\forall \lambda \in \mathsf{N}, \forall K$ generated using $KeyGen(1^\lambda)$ and all sequences of add, delete and search operations on γ, every search operation outputs the correct set of file identifiers, except with a negligible probability.

Security: In the ideal scenario, a DSSE scheme should have the following two properties.

1. From $(\gamma, \mathbf{c})$ given initially, the server should not learn any information about files in $\mathbf{f}$.
2. From a sequence of tokens (search, add or delete) $t_1, t_2, \ldots, t_n$, the server should learn nothing about corresponding queries and file collection $\mathbf{f}$.

Goldreich [41] first introduced an oblivious RAM (ORAM)-based search scheme that had these properties. However, this scheme is not practically implementable due to its high computation complexity.

Definition 4.2 (CKA2-security of a DSSE scheme)**.** *[48] Let* $\Sigma = (KeyGen,\ Enc,\ AddToken,\ Add,\ Build,\ SearchToken,\ Search,\ DeleteToken,\ Delete,\ Dec)$ *be an inverted index-based DSSE scheme. Let* $\mathcal{A}$ *be a stateful adversary,* $\mathcal{C}$ *be a challenger,* $\mathcal{S}$ *be a stateful simulator and* $\mathcal{L} = (\mathcal{L}_{bld}, \mathcal{L}_{srch}, \mathcal{L}_{add}, \mathcal{L}_{del})$ *be a stateful leakage algorithm. Let us consider the following two games.*

Real$_{\mathcal{A}}(\lambda)$*:*

1. *The challenger $\mathcal{C}$ generates a key $K \leftarrow Gen(1^\lambda)$.*
2. *$\mathcal{A}$ generates a set of files $\mathbf{f}$ and sends it to $\mathcal{C}$.*
3. *$\mathcal{C}$ computes $(\gamma, \boldsymbol{c}) \leftarrow Build(K, \mathbf{f})$ and sends $(\gamma, \boldsymbol{c})$ to $\mathcal{A}$*
4. *$\mathcal{A}$ makes a polynomial number of adaptive queries. In each query $\mathcal{A}$ sends either a search query for a keyword w or an add query for a file f_1 or a delete query for a file f_2 to $\mathcal{C}$.*
5. *Depending on the query, $\mathcal{C}$ returns either the search token $t_s \leftarrow SearchToken(K, w)$ or the add token $t_a \leftarrow AddToken(K, f_1)$ or the delete token $t_d \leftarrow DelToken(K, f_2)$ to $\mathcal{A}$.*
6. *Finally $\mathcal{A}$ returns a bit b that is output by the experiment.*

Ideal$_{\mathcal{A},\mathcal{S}}(\lambda)$*:*

1. *$\mathcal{A}$ generates a set of files $\mathbf{f}$. It gives $\mathbf{f}$ and $\mathcal{L}_{bld}(\boldsymbol{f})$ to $\mathcal{S}$.*
2. *On receiving $\mathcal{L}_{bld}(\mathbf{f})$, $\mathcal{S}$ generates $(\gamma, \boldsymbol{c})$ and sends it to $\mathcal{A}$*
3. *$\mathcal{A}$ makes a polynomial number of adaptive queries $q \in \{w, f_1, f_2\}$. For each query, $\mathcal{S}$ is given either $\mathcal{L}_{srch}(w, \mathbf{f})$ or $\mathcal{L}_{add}(f_1, \mathbf{f})$ or $\mathcal{L}_{del}(f_2, \mathbf{f})$.*
4. *Depending on the query q, $\mathcal{S}$ returns to $\mathcal{A}$ either search token t_s or add token t_a or delete token t_d.*
5. *Finally $\mathcal{A}$ returns a bit b that is output by the experiment.*

We say Σ is $\mathcal{L}$-secure against adaptive dynamic chosen-keyword attacks if for any PPT (probabilistic polynomial-time) adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that

$$|Pr[\textbf{\textit{Real}}_{\mathcal{A}}(\lambda) = 1] - Pr[\textbf{\textit{Ideal}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mu(\lambda) \tag{4.1}$$

4.2 Our proposed scheme Trids

Scheme overview: We use a one-dimensional array A and two hash tables T and T'. The client starts with a collection of documents $\mathbf{f} = \{f_1, f_2, \ldots, f_n\}$. A tri-directional linked list (TDL) is created taking the cells from A as shown in Fig. 4-1. Each entry in T corresponds to a keyword $w \in W$ and that in T' corresponds to a file $f \in \mathbf{f}$.

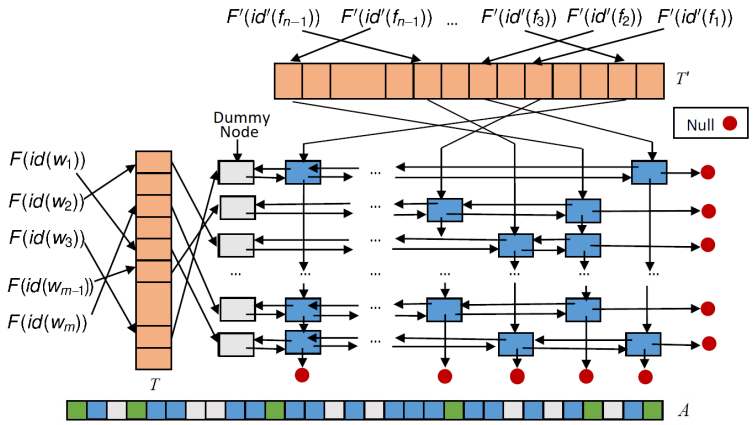


Figure 4-1: Internal data structure of TDL

Each node in the TDL has pointers to three directions: right, down, and left. Right pointers are used for searching while down pointers are used during the deletion of a file and left pointers for modifying the list after deletion.

The client takes a set of files and builds an encrypted inverted index. The index together with encrypted files are sent to the cloud server. Later, the client generates tokens for search, add or delete query, and the cloud server returns the corresponding results (if any) after processing the queries.

In order to construct the index, for each keyword, the number of files containing the keyword is calculated. The same number of cells are chosen at random from A. A TDL is created with the chosen cells and each node is filled with a file identifier. A dummy node is then added to the beginning of the linked list. Finally, a pointer to the dummy node is kept in the row of T indexed by the keyword. Nodes belonging to the same file identifier are linked with down pointers. The head pointer of that list is kept in the table T' corresponding to the position of the file in encrypted form. However, this linking is done simultaneously while building the TDL.

A search token consists of an index in T, its decryption key, and some extra information. This extra information helps the cloud to decrypt the nodes. Similarly, for adding or deleting, the position of the file in T', its decryption keys are sent with the corresponding token. The cloud server traverses through the TDL and modifies it (if required) as per the requested token.

4.2.1 Description of our proposed scheme Trids

Our main idea is to construct a DSSE scheme based on an efficient inverted index. The efficiency of our scheme is compared with existing schemes in Section 4.3. In the beginning, an encrypted

inverted index corresponding to a set of documents is built and then uploaded to the cloud server. Later, on request of the client, the cloud server performs a search or add or delete on the index and returns the results. It updates the index as per client requests.

Our proposed scheme Trids consists of the algorithms $KeyGen()$, $Enc()$, $Build()$, $AddToken()$, $Add()$, $SearchToken()$, $Search()$, $DeleteToken()$, $Delete()$ and $Dec()$. For file encryption, a CPA-secure[1] symmetric encryption scheme $SE = (SE_{gen}, SE_{enc}, SE_{dec})$ is used.

In $KeyGen()$ (see Algo. 6), a secret key of length λ is generated. The initial inverted index γ is built using $Build()$ function corresponding to a set of files **f** and a secret key K. γ is a tri-directional linked list (TDL) which is built with the help of two hash tables T and T' and an one dimensional array A of structures of size $|W|$, l_{max} and $l_w.l_{max}$[2] respectively.

Algorithm 6: $KeyGen(\lambda)$

1 $k_1, k_1', k_2, k_2', k_3 \xleftarrow{\$} \{0,1\}^\lambda$
2 $k \leftarrow SE_{gen}(1^\lambda)$
3 $K = (k, k_1, k_1', k_2, k_2', k_3)$
4 **return** K

The array A is an array of structures. Each element of A is said to be a cell that is used in TDL. Each node of the TDL is a 6-tuple ($\overline{L}$, $\overline{R}$, $\overline{D}$, $\overline{Rs}$, $\overline{id'}(f)$, r) where $\overline{L} = L \oplus H_1(K_f, r)$, $\overline{R} = R \oplus H_2(K_f, r)$, $\overline{D} = D \oplus H_3(K_f, r)$, $\overline{Rs} = Rs \oplus H_4(K_w, r)$ and $\overline{id'}(f) = id'(f) \oplus H_5(K_w, r)$. L, R, D and Rs are the addresses of the nodes on the left, right, down, and right respectively. These addresses are not disk memory locations but the position indices in A. For example, the left pointer L indicates the $(L+1)$th entry of A, i.e., $A[L]$. $id'(f)$ is the identity of the file f. r is a random number of length λ. R and Rs have the same value while $\overline{R}$ and $\overline{Rs}$ are different. The address Rs is used for searching on TDL while L, R, and D are used for updating a file. K_f and K_w are the keys corresponding to the file f and the keyword w respectively.

For an integer i in range $|A|$, for writing simplicity, we denote $A[i].\overline{L}$, $A[i].\overline{R}$, $A[i].\overline{D}$, $A[i].\overline{Rs}$, $A[i].\overline{id'}(f)$, and $A[i].r$ as $A[i][0]$, $A[i][1]$, $A[i][2]$, $A[i][3]$, $A[i][4]$, and $A[i][5]$ respectively.

It can be observed that if we consider only right pointers corresponding to a keyword w, then it visits the files which contain the keyword w. On the other hand, if we traverse through only using down pointers corresponding to a file f, we visit all keywords of f.

Let W be the set of all keywords. T (or T') is a hash table of length $|W|$ (or l_{max}). A function F (or F') maps each keyword (file) identifier $id(w)$ (or $id'(f)$) to a distinct entry of T (or T'). Thus, $T[F(id(w))]$ (or $T'[F'(id'(f))]$) is the entry corresponding to w (or f) which stores the head of the linked list for w (or f). A secret key is used by the client in id and id' to make functions private. It prevents the cloud server to learn about a keyword/file name from its identifiers.

[1] CPA-security is defined as indistinguishability under chosen plaintext attack or IND-CPA [10]

[2] l_w is the average number of distinct keywords in a file.

There is a *dummy node* in front of each linked list corresponding to each keyword w (see Fig. 4-1). For example, for a keyword w, a random node is taken and its address is kept in $T[F(id(w))]$ encrypted using $G(id(w))$. The node is dummy because it does not keep any file identifier stored in it. While deleting the lastly added file, it helps not to leak information about the keywords not searched before. A dummy node is added to the linked list for every keyword when it is added for the first time in the database. Dummy nodes added during $Build()$ process are chosen randomly from the empty cells, and those added during $Add()$ are chosen from L_{free}, a list of free cells from A.

A free list L_{free} is a sequence of random nodes from A where the first node of the sequence is $A[0]$. In the list, each node keeps the address of the next node. The selection of randomness is done by the client. The list is at the end of $Build()$ process. Algorithm $L_{free}.in()$ (see Algo. 15) and $L_{free}.out$ (see Algo. 14) are the only operations defined to access the list. When a file is deleted, the deleted nodes are added to the list using $L_{free}.in()$ and at the time of addition of a new file, new nodes are taken from L_{free} using $L_{free}.out$.

Initially, the client passes a set of files **f** and a secret key K to the $Build()$ function (see Algo. 7). Then an inverted index γ is initialized with A, T and T'. For each $w \in W$, a linked list is made with the randomly chosen cells from A. Each entry corresponding to a keyword w stores the identifiers of the files that contain the keyword w. The file identifiers are kept in an encrypted form. At the end of $Build()$ process the empty cells in A are chosen at random and linked to form a singly linked list. The first pointer of the list is kept in $A[0]$. This list is the free list and denoted by L_{free}. Finally, the updated γ, together with the set of encrypted files **c**, is uploaded to the cloud server by the client.

We see that before building the encrypted index, the client requires some computations. We assume that the client uses a large computing device for this. Since the computation is a one-time process, the client can do it easily.

To add a file f, an add token t_a is generated using $AddToken()$ (see Algo. 8). The client sends t_a together with the encrypted version c of the file f. File encryption is done using $Enc()$ (which is $SE_{enc}()$). $Dec()$ (i.e., $SE_{dec}()$) is used later to decrypt retrieved encrypted files. γ is updated with every call of $Add()$ (see Algo. 9). When add token is received by $Add()$, the set of keyword identifiers are then sorted.

For each keyword w in $\hat{f}$, an unused node is taken from L_{free} and added in front of the dummy node of the linked list corresponding to w. If there is no dummy node, a node is taken from L_{free} and is made as a dummy node. Then another node corresponding to the keyword is taken and added. Thus, entries of the table T and dummy nodes are updated. All new entries are also linked

Algorithm 7: $Build(K, \mathbf{f})$

$\mathbf{c} \leftarrow \phi$
Allocate T, T' and A of size $|W|$, l_{max} and $l_w.l_{max}$
$\gamma \leftarrow (A, T, T')$
for *each file* $f \in \boldsymbol{f}$ **do**
  $\hat{f} \leftarrow$ set of distinct keywords $\in f$
  $sf \leftarrow \{id(w) : w \in \hat{f}\}$ where the set is sorted
  Choose $|\hat{f}|$ unused cells from A
  Store $id'(f)$ in the chosen cells
  Make downward linked list
  Add the linked list in TDL
  If a keyword w appears first time add dummy nodes and update corresponding entry at $T[F(id(w))]$.
  Encrypt each node according to the structure
  Keep the first node of the downward linked list in $T'[F'(id'(f))]$ encrypted with $G'(K[4], id'(f))$.
end
Choose unused cells of A and make free list L_{free}
Keep head of the linked list at $A[0]$.
return $(\mathbf{c}, \gamma)$

Algorithm 8: $AddToken(K, f)$

Parse $(k, k_1, k_1', k_2, k_2', k_3) \leftarrow K$
$\hat{f} \leftarrow \{w_1, w_2, \ldots, w_{|\hat{f}|}\}$, set of distinct keywords in f.
$sf \leftarrow \{id(w) : w \in \hat{f}\}$ where the set is sorted
$k_f \leftarrow P(k_3, id'(f))$
for $id(w_i) \in sf$ **do**
  $r_i \xleftarrow{\$} \{0,1\}^{\lambda}$; $k_{w_i} \leftarrow P(k_3, id(w))$
  $h_1 \leftarrow H_1(k_f, r_i)$; $h_2 \leftarrow H_2(k_f, r_i)$
  $h_3 \leftarrow H_3(k_f, r_i)$
  $h_4 \leftarrow H_4(k_{w_i}, r_i)$; $h_5 \leftarrow id'(f) \oplus H_5(k_{w_i}, r_i)$
  $p_i \leftarrow (F_{k_1}(id(w_i)), G_{k_2}(id(w_i)), h_1, h_2, h_3, h_4, h_5, r_i)$
end
$t_a \leftarrow (F'_{k_1'}(f), G'_{k_2'}(f), p_1, p_2, \ldots, p_{|\hat{f}|})$
return t_a

Algorithm 9: $Add(t_a, c, \mathbf{c}, \gamma)$

$(A, T, T') \leftarrow \gamma$
for $i = 0$ ***to*** $|\hat{f}|$ **do**
  $N \leftarrow L_{free}.out()$
  $N_2 \leftarrow T[p_i[0]] \oplus p_i[1]$
  $N_1 \leftarrow A[N_2][3] \oplus p_i[1]$
  $A[N][0] \leftarrow N_2 \oplus p_i[2]$
  $A[N][1] \leftarrow p_i[3]$; $A[N][3] \leftarrow p_i[5]$
  $A[N][4] \leftarrow p_i[6]$; $A[N][5] \leftarrow p_i[7]$
  $A[N_2][3] \leftarrow A[N_2][3] \oplus N1 \oplus N_2$
  if $N_1! = 0$ **then**
    $A[N_1][0] \leftarrow A[N_1][0] \oplus N \oplus N_2$
  end
  if $i = 0$ **then**
    $T'[t_a[0]] \leftarrow N \oplus t_a[1]$
  end
  else
    $A[N'][2] \leftarrow A[N'][2] \oplus N$
  end
  $A[N][2] \leftarrow p_i[4]$
  $N' \leftarrow N$
end
$\gamma' \leftarrow (A, T, T')$; $\mathbf{c}' \leftarrow \{c\} \cup \mathbf{c}$
return $(\mathbf{c}', \gamma')$

Algorithm 10: $SearchToken(K, w)$

Parse $(k, k_1, k_1', k_2, k_2', k_3) \leftarrow K$
$k_w \leftarrow P(k_3, id(w))$
$t_s \leftarrow (F_{k_1}(id(w)), G_{k_2}(id(w)), k_w)$
return t_s

Algorithm 11: $Search(t_s, \gamma)$

$(A, T, T') \leftarrow \gamma$
$ptr \leftarrow T[t_s[0]] \oplus t_s[1]$
$I_w \leftarrow \{\}$
while $ptr != Null$ **do**
  $N \leftarrow ptr$
  $ptr \leftarrow H_4(t_s[2], A[N][5]) \oplus A[N][3]$
  $b \leftarrow H_5(t_s[2], A[N][5]) \oplus A[N][4]$
  if $A[N]$ *is not a dummy node* **then**
    $I_w \leftarrow I_w \cup \{b\}$
  end
end
return I_w

Algorithm 12: $DeleteToken(K, f)$

Parse $(k, k_1, k_1', k_2, k_2', k_3) \leftarrow K$
$k_f \leftarrow P(k_3, id'(f))$
$t_d \leftarrow (F'_{k_1'}(id'(f)), G'_{k_2'}(id'(f)), k_f)$
return t_d

Algorithm 13: $Delete(t_d, \gamma, \mathbf{c})$

$(A, T, T') \leftarrow \gamma$
$N \leftarrow T'[t_d[0]] \oplus t_d[1]$
$idf \leftarrow H_4(t_d[2], A[N][5]) \oplus A[N][4]$
while $N! = Null$ **do**
  $r \leftarrow A[N][5]$
  $L \leftarrow A[N][0] \oplus H_1(t_d[2], r)$
  $R \leftarrow A[N][1] \oplus H_2(t_d[2], r)$
  $D \leftarrow A[N][2] \oplus H_3(t_d[2], r)$
  if $R! = Null$ **then**
    $A[R][0] \leftarrow A[R][0] \oplus N \oplus L$
  end
  $A[L][1] \leftarrow A[L][1] \oplus N \oplus R$
  $A[L][3] \leftarrow A[L][3] \oplus N \oplus R$
  Fill $A[N]$ with random string
  $L_{free}.in(N)$
  $N \leftarrow D$
end
$T'[t_d[0]] \leftarrow t_d[1]$
$\gamma' \leftarrow (A, T, T')$
$\mathbf{c}' \leftarrow \mathbf{c}$ after deleting file pointed by idf
return $(\gamma', \mathbf{c}')$

Algorithm 14: $L_{free}.out()$

$N \leftarrow A[0][3]$
if $N == 0$ **then**
  return 0
end
$A[0][3] \leftarrow A[N][3]$
return N

Algorithm 15: $L_{free}.in(N)$

$h \leftarrow A[0][3]$
$A[0][3] \leftarrow N$
$A[N][3] \leftarrow h$
return 1

downward whose first entry is kept in $T'[F'(id'(f))]$. Fig. 4-2 shows the addition of a node while adding a file. Note that while adding a file, a node corresponding to the keyword is always added just after the dummy node.

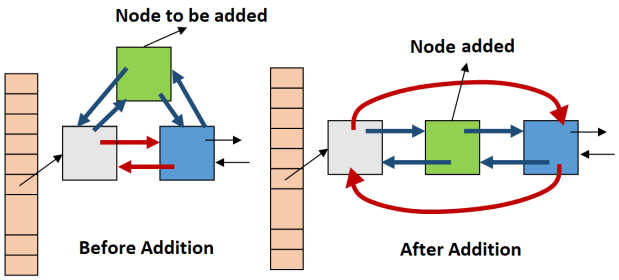


Figure 4-2: Addition of a node in TDL

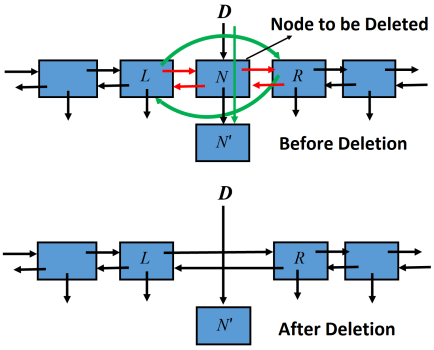


Figure 4-3: Deletion of a node in TDL

To search the files containing a keyword w, the client generates a search token t_s using $SearchToken()$ (see Algo. 10) and sends it to the cloud server. The search token contains the position $F(id(w))$ in the table T and its decryption key $G(id(w))$. So, the cloud server can find the head of the linked list. However, with the help of k_w, the cloud server can traverse the list and decrypt the file identifiers as described in Algo. 11. Finally, the cloud server returns the set of identifiers of the files containing w using $Search()$.

To delete a file, the client sends $t_d = (F'_{k'_1}(id'(f)),\ G'_{k'_2}(id'(f)),\ k_f)$, the delete token generated using $DeleteToken()$ (see Algo. 12) to the cloud server. In Fig. 4-1, it can be seen that nodes corresponding to a file are in a linked list directed downwards. After receiving the delete token the cloud server runs $Delete()$ (see Algo. 13). It finds head of that linked list in $T'[(F'_{k'_1}(id'(f))]$ by decrypting using $G'_{k'_2}(id'(f))$.

Then the cloud server traverses through the linked list downwards and adds each node to the free list L_{free} (see Algo. 15). Before adding to L_{free}, the content of the node is replaced by some random values. While making each node free, its right and left nodes in TDL are also updated. Fig. 4-3 shows the deletion of a node where red arrows show new linking after deletion.

Why dummy nodes are required? If there are no dummy nodes, then the addresses of the new nodes will be directly added to T. After uploading the inverted index to the cloud server, if the client sends a query to delete the last added file, the cloud server has to update corresponding entries in T as the first node has been changed. Keeping dummy nodes prevents the cloud server to get any information about table T while deleting. The pointers in dummy nodes are updated instead of entries in T while deleting the last added file.

4.2.2 Example

We illustrate our scheme with an example. Let T, T' and A be of length 6, 10 and 20 respectively. Let $W = \{w_1, w_2, w_3, w_4, w_5\}$ and $\mathbf{f} = \{f_1, f_2, f_3\}$. Let $F(id(w_1)) = 3$, $F(id(w_2)) = 0$, $F(id(w_3)) = 4$, $F(id(w_4)) = 2$, $F(id(w_5)) = 5$, $F'(id'(f_1)) = 5$, $F'(id'(f_2)) = 3$, $F'(id'(f_3)) = 7$. $\hat{f}_1 = \{w_1, w_3, w_4\}$, $\hat{f}_2 = \{w_2, w_3\}$ and $\hat{f}_3 = \{w_1, w_2, w_4\}$.

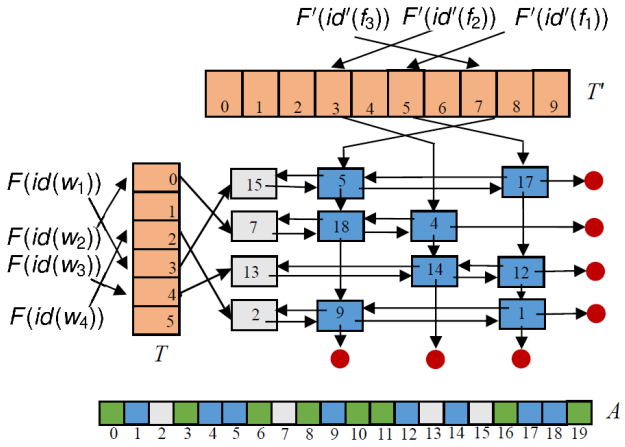


Figure 4-4: Example of the TDL built with W and $\mathbf{f}$

In Build process, let the set of locations $\{17, 12, 1\}$, $\{4, 14\}$ and $\{5, 18, 9\}$ are selected from the array A for the files f_1, f_2 and f_3 respectively. Addresses 15, 7, 13 and 2 of A are selected randomly for dummy nodes corresponding to the keywords w_1, w_2, w_3 and w_4 respectively.

Fig. 4-4 shows structure after $Build()$. Color blue indicates the cells are chosen for a keyword-file pair, gray indicates dummy nodes. For example, 18 has been chosen for (w_2, f_3) and 7 for w_2. Green indicates remaining cells which are used to make free list L_{free} which is as follows. $A[0][3] = 8$, $A[8][3] = 11$, $A[11][3] = 16$, $A[16][3] = 19$, $A[19][3] = 3$, $A[3][3] = 10$.

Thus, entries corresponding to the keywords in table T are $T[F(id(w_1))] = 15 \oplus G(id(w_1))$, $T[F(id(w_2))] = 7 \oplus G(id(w_2))$, $T[F(id(w_3))] = 13 \oplus G(id(w_3))$, $T[F(id(w_4))] = 2 \oplus G(id(w_4))$. Similarly entries of T' are $T'[F(id'(f_1))] = 17 \oplus G'(id'(f_1))$, $T'[F(id'(f_2))] = 4 \oplus G'(id'(f_2))$ and $T'[F(id'(f_3))] = 5 \oplus G'(id'(f_3))$.

If we add a file $f_4 = \{w_1, w_3, w_5\}$, then there will be a new entry for the new keyword w_5 in Table T as $T[F(id(w_5))] = 6 \oplus G(id(w_5))$, and the entries of the free list becomes $A[0][3] = 19$, $A[19][3] = 3$, $A[3][3] = 10$. Fig. 4-5 shows the changed structure after addition. There will be a new entry in T' as $T'[F(id'(f_4))] = 8 \oplus G'(id'(f_4))$.

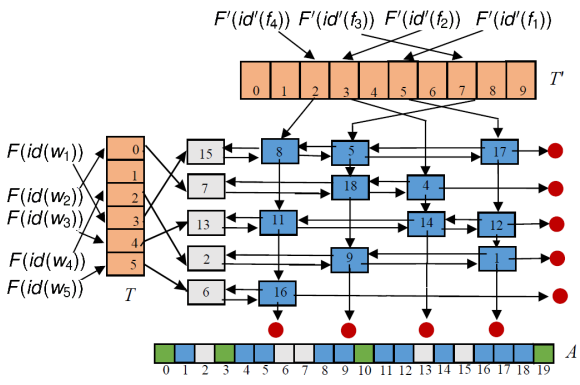


Figure 4-5: Example of an addition of a file

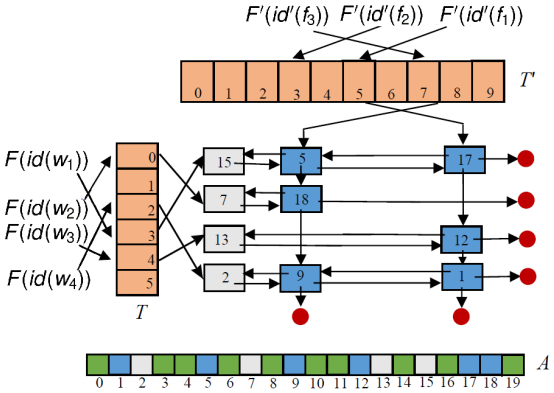


Figure 4-6: Example of a deletion of a file

The deletion of a file does not affect T. The links of the nodes corresponding to the file being deleted are changed and the entry in T' becomes null. Then, the deleted nodes are added to the free list. Fig. 4-6 shows the deletion of f_2 from the built index (see Fig.4-4).

4.3 Comparison of Trids with existing DSSE schemes

We have used a TDL to design our scheme. It consists of an array A and two hash tables T and T'. We now compare our proposed scheme with some well known DSSE schemes and show that the design of our scheme gives better results in many directions. The comparison is summarized in Table 4.2 and Table 4.3.

Inverted Index-Based DSSE Kamara et al. [48] first presented a DSSE scheme that uses the concept of the inverted index with linked lists. They used two arrays and two hash tables where we have used only one array. Let a be the size of an address of A, b be the size of a file identifier and random numbers be of λ-bits. Then, [48] requires $(7a + 2b + 2\lambda)$-bits storage corresponding to each cell. For our case, we require only $(4a + b + \lambda)$-bits. Thus, we have approximately halved the storage requirement for the encrypted index.

Secondly, instead of accessing two tables simultaneously, accessing only one table reduces the access cost, especially when the tables reside in different storage locations. Thus, in our case, the search is faster.

Finally, compared to [48], our scheme decreases the leakage $\mathcal{L}_{del}$ (see Eqn. 4.2) while deletion. For each entry corresponding to the keywords belonging to the deleted file, [48] reveals file ids stored in both left and right cells and also reveals that these two files have a common keyword. In

Table 4.2: Comparison among DSSE schemes based on client-side costs

Scheme	Communication bandwidth†			Computation			Client storage
	Search	Add	Delete	Search	Add	Delete	
HK [42]	$O(1)$	$O(\lvert\hat{f}\rvert)$	$O(1)$	$O(1)$	$O(\lvert\hat{f}\rvert)$	$O(1)$	$O(N)$
SPS [87]	$O(\lvert f_w\rvert + \log N)$ rounds*	$O(\log N)$ rounds*	$O(\log N)$ rounds*	$O(4.min\{\alpha + \log N, \lvert f_w\rvert \log^3 N\})$	$O(\lvert\hat{f}\rvert . \log^2 N)$	$O(\lvert\hat{f}\rvert \log^2 N)$	$O(N^\beta)$
Bost [16]	$O(1)$	$O(f_w)$ rounds*	-	$O(1)$	$O(4\lvert\hat{f}\rvert)$	-	$O(\lvert W\rvert(\log \lvert\mathbf{f}\rvert + \lambda))$
KPR [48]	$O(1)$	$O(\lvert\hat{f}\rvert)$	$O(1)$	$O(1)$	$O(\lvert\hat{f}\rvert)$	$O(1)$	$O(1)$
Our scheme	$O(1)$	$O(\lvert\hat{f}\rvert)$	$O(1)$	$O(1)$	$O(\lvert\hat{f}\rvert)$	$O(1)$	$O(1)$

$N \equiv$ number of keyword-file pairs, $W \equiv$ set of all possible keywords, $f_w \equiv$ set of files containing a keyword w, $\hat{f} \equiv$ set of distinct keywords in f, $\mathbf{f} \equiv$ set of files in the database, $|.| \equiv$ cardinality, $0 < \beta < 1$, $a_w \equiv$ # times the queried keyword w was historically added to the database, $r \equiv$ one ORAM read in TWORAM, $d_w \equiv$ the number of times the searched for keyword has been added or deleted, $p \equiv$ # processors, $\bar{O} \equiv$ order avoiding $\log\log N$, computation includes ORAM access. $a \equiv$ length of keyword-file storage, $b \equiv$ max. supported length of file id, $\lambda =$ security parameter.

† This amount of bandwidth (measured per keyword-file storage) is required by the client to request the cloud server to perform different tasks.
* Communication bandwidth is very high due to the large number of communication rounds needed.

our case, the deletion of a file only leaks information about the file identifiers stored in immediate neighboring nodes if they are previously queried. Other leakages are the same for both.

History-Based DSSE Hahn and Kerschbaum [42] presented a DSSE scheme based on search history. They used two types of indices: regular index[3] and inverted index. Initially, the regular index is constructed and an entry corresponding to a keyword is added into the inverted index on the first search of the keyword.

This scheme has asymptotically optimal search time for the long-running system. Initial search time is linear in twice the total number of stored keywords (it is counted twice if a keyword belongs to a file). Subsequent search times are close to constant. In our scheme, all searches take constant time in the number of keywords, and it is linear in the number of files it belongs to. Thus, [42] performs better for a particular search only if it has $\sigma(|W|^2)$ repetitions.

Blind Storage-Based DSSE Naveed et al. [72] proposed Blind Storage and implemented a DSSE scheme via Blind Storage. We recall the scheme is described in Section 3.1.2 of Chapter 3. Instead of cloud computation, their scheme is based on a cloud storage service. The cloud server only keeps data uploaded to it and sends specific stored data as requested by the client.

[3] A regular index contains a table of files and the corresponding keywords for each file.

Table 4.3: Comparison among DSSE schemes based on server-side costs

Scheme	Computation			Storage
	Search	Add	Delete	Index size
HK [42]	$O(N)$ for first search $O(1)$ for subsequent search	$O(\|\hat{f}\|)$	$O(\|W\|)$	$O(N.(a+2b))$
SPS [87]	-	-	-	$O(N.(3a+b))$
Bost [16]	$O(2\|f_w\|)$	$O(\|\hat{f}\|)$	-	$O(N)$
KPR [48]	$O(\|f_w\|)$	$O(\|\hat{f}\|)$	$O(\|\hat{f}\|)$	$O(N.(7a+2b+2\lambda))$
Our scheme	$O(\|f_w\|)$	$O(\|\hat{f}\|)$	$O(\|\hat{f}\|)$	$O((N+\|W\|).(4a+b+\lambda))$

However, the scheme has higher communication and storage costs. It requires several rounds of communication for most of the queries. In our scheme, each query requires only one round of communication. Secondly, in their scheme, all computations are done by the client. In our scheme, most of the computations, except the token generation, are done at the server-side.

Oblivious-sort Based DSSE Stefanov et al. [87] proposed a DSSE scheme using oblivious sorting. They reduced the leakage by making the scheme forward private. However, [87] requires RAM as large as the size of the database. Moreover, it has a search complexity of $O(|\hat{f}|.\log^3 N)$ with $O(\log N)$ rounds of communication, and it needs client storage $O(N)$. Our scheme has $O(|\hat{f}|)$ search complexity and does not require a RAM of such a large size. However, a larger RAM in our scheme can process a query faster.

Sophos Bost [16] proposed a forward private scheme *Sophos*-B that relies on trapdoor permutations. However, the scheme does not support delete operations although *Sophos* can be extended to support deletion at the cost of storage and computation. However, our scheme supports efficient deletion.

Others Recent forward and backward private DSSE schemes, [18, 90, 25], require significant amount of client storage (at least $O(|W|)$) which makes the schemes inappropriate for lightweight clients.

4.4 Security Analysis

In this section, we discuss different types of leakages that are made due to the initial database uploads and query tokens. After building the initial encrypted database with the inverted index, the client uploads both of them to the cloud server except the secret key K. Let $\mathcal{L}_{bld}$, $\mathcal{L}_{srch}$, $\mathcal{L}_{add}$, $\mathcal{L}_{del}$

be the leakages associated with build, search, add and delete respectively. We define the leakage function $\mathcal{L}$ as a 4-tuple of the above leakages, i.e., $\mathcal{L} \equiv (\mathcal{L}_{bld}, \mathcal{L}_{srch}, \mathcal{L}_{add}, \mathcal{L}_{del})$.

Description of leakage function $\mathcal{L}$: We define leakage functions as follows.

- From the initial database and encrypted inverted-index, the cloud server can get the size of the array A and each encrypted file, the set of all possible keyword identifiers, set of all possible file identifiers. Since the free list L_{free} is kept in an unencrypted form, it is also revealed to the cloud server. We define $\mathcal{L}_{bld}$ as follows.

$$\mathcal{L}_{bld}(\mathbf{f}) = \{|A|, |T|, |T'|, \{|c_f| : f \in \mathbf{f}\}, \{id(w) : w \in W\}, \{id'(f) : f \in \mathbf{f}\}, L_{free}\}$$

- When a client sends a search query token to the cloud server for a keyword w and the cloud server executes the query, it gets the identifier of w as well as the set of identifiers of the files that contain w. We define search leakage $\mathcal{L}_{srch}$ as $\mathcal{L}_{srch}(w, \mathbf{f}) = \{id(w), I_w\}$, where $I_w = \{id'(f) : w \in f \wedge f \in \mathbf{f}\}$, i.e., I_w is the set of identifiers of the files containing w.

- From an addition query, the cloud server gets information about identity of the added file and set of identifiers of the keywords belonging to the file. It also gets the information if the keyword identifiers are previously searched or added. Leakage from addition $\mathcal{L}_{add}$ is defined as

$$\mathcal{L}_{add}(f, \mathbf{f}) = \{id'(f), \{(id(w), apprs(w))\}_{w \in \hat{f}}, L_{free_{add}}\}$$

  where $apprs(w) = 1$ if w ($\in \hat{f}$) appeared in any previous add or search query, otherwise $apprs(w) = 0$, i.e., $apprs(w) = 1$ if $w \in Q_{sa}$ and 0 otherwise. By Q_{sa}, we denote the set of $id(w)$'s which are previously searched or added. $L_{free_{add}}$ is the sequence of indices taken from L_{free} at the time of addition. Note that, in the leakage, the keyword identifiers are there, not the keywords.

- From deletion of a file, the cloud server only gets the identifier of the deleted file, length of the deleted file and the set of identifiers of those keywords of the deleted file that were previously searched. The leakage from deletion $\mathcal{L}_{del}$ is defined as

$$\mathcal{L}_{del}(f, \mathbf{f}) = \{id'(f), |\hat{f}|, acc(w), L_{free_{del}}\} \tag{4.2}$$

  where $acc(w) = \{id(w) : w \in \hat{f} \cap Q_s\}$, By Q_s, we denote the set of $id(w)$'s which are previously searched only. $L_{free_{del}}$ is the set of indices added to L_{free} at the time of deletion.

Theorem 4.1. *If SE is CPA-secure and F, F' are PRP, G, G' and P are PRG, then the DSSE scheme Σ is $\mathcal{L}$-secure against adaptive dynamic chosen-keyword attack.*

Proof. To prove the above theorem, it is sufficient to show that there exists a simulator $\mathcal{S}$ such that for every PPT (probabilistic polynomial-time) adversary $\mathcal{A}$, the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ are computationally indistinguishable. We will construct such a simulator $\mathcal{S}$ which adaptively simulates an encrypted inverted index $\widetilde{\gamma} = (\widetilde{A}, \widetilde{T}, \widetilde{T'})$, a sequence of simulated ciphertexts $\widetilde{\mathbf{c}}$, a sequence of $poly(\lambda)$ simulated queries $q = (q_1, q_2, \ldots)$ and a sequence of $poly(\lambda)$ simulated tokens $\widetilde{t_1}, \widetilde{t_2}, \ldots$ (where $\widetilde{t_i}$ is either a search or add or delete token) as follows.

- (Set up data structures) Given $\mathcal{L}_{bld}(\mathbf{f})$

  1. *Key generation*: $\mathcal{S}$ first generates $k \leftarrow SE_{gen}(1^{\lambda})$ for encrypting files. Then, $\forall i \in [|W|]$, it randomly chooses a λ-bit key $K_{id(w_i)}$, associated with keyword identifier $id(w_i)$. Then, $\forall j \in [|\mathbf{f}|]$, it chooses a λ-bit key $K_{id'(f_j)}$, uniformly at random, associated with file identifier $id'(f_j)$.
  2. *Simulate ciphertexts*: For all $f \in \mathbf{f}$, $\mathcal{S}$ computes $\widetilde{c_f} \leftarrow SE_{enc}(k, 0^{|f|})$. Then, it sets $\widetilde{\mathbf{c}} \leftarrow \{\widetilde{c_f} : f \in \mathbf{f}\}$
  3. *Simulate A*: $\mathcal{S}$ generates an array $\widetilde{A}$ of size $|A|$. It fills each cell of $\widetilde{A}$ with random bit-strings of the form (L, R, D, Rs, idf, r) where L, R, D and Rs are of length $\log|A|$, idf is of length $\log|\mathbf{c}|$ and r is of length λ. Then marks all empty cells in $\widetilde{A}$ as unused.
  4. *Simulate T*: $\mathcal{S}$ generates a dictionary $\widetilde{T}$ of size $|W|$, and for all $w \in W$, $\mathcal{S}$ stores a random bit-string v_w of length $\log|A|$ as $\widetilde{T}[id(w)] = v_w$. Then it copies $\widetilde{T}$ in $\widetilde{T}_c$.
  5. *Simulate T'*: $\mathcal{S}$ generates a dictionary $\widetilde{T'}$ of size $|\mathbf{f}|$. Then, in each entry of $\widetilde{T'}$, $\mathcal{S}$ stores a random bit-string of length $\log l_{max}$.
  6. Make $\widetilde{L}_{free}$ in $\widetilde{A}$ as L_{free}. Here $\mathcal{S}$ generates a free list according to the given leakage.

- (Simulating search token t_s) Given $\mathcal{L}_{srch}(w, \mathbf{f})$, $\mathcal{S}$ checks whether $id(w)$ has appeared before in the previous leakages. It takes a set Q_{sa} (initially empty) of $id(w)$'s which are previously searched or added.

Case 1: If $id(w) \notin Q_{sa}$,

If ($I_w = \phi$), $\mathcal{S}$ selects an unused cell index $\alpha_0 \in [|\widetilde{A}|]$, takes $\alpha_1 \leftarrow \perp$ and links the chosen cell as $\widetilde{T}[id(w)] = \alpha_0$ and $\widetilde{A}[\alpha_0][1] = \perp$. Moreover, $\widetilde{A}[\alpha_0]$ is marked as the dummy node for $id(w)$.

Else if ($I_w \neq \phi$), $\mathcal{S}$ selects $|I_w|+1$ number of unused cell indices $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{|I_w|} \in [|\widetilde{A}|]$ at random and mark them for $id(w)$. File identifiers from the leakage are stored in $\widetilde{A}[\alpha_i]$'s for $i = 1(1)n$ and $\widetilde{A}[\alpha_0]$ is marked as the dummy node. The cells are then linked as

- $\widetilde{T}[id(w)] \leftarrow \alpha_0$
- $\widetilde{A}[\alpha_i][1] \leftarrow \alpha_{i+1},\ 0 \leq i < |I_w|,\ \widetilde{A}[\alpha_{|I_w|}][1] \leftarrow \perp$
- $\widetilde{A}[\alpha_i][0] \leftarrow \alpha_{i-1},\ 0 < i \leq |I_w|,\ \widetilde{A}[\alpha_0][0] \leftarrow \perp$

Case 2: If $id(w) \in Q_{sa}$,

1. If $\widetilde{A}[\widetilde{T}[id(w)]][1] = \perp$, then $id(w)$ is previously searched but is not present in any file. For this case, $\alpha_0 \leftarrow \widetilde{T}[id(w)]$ and $\alpha_1 \leftarrow \perp$.
2. If $\widetilde{A}[\widetilde{T}[id(w)]][1] = \alpha_1 (\neq \perp)$, $\mathcal{S}$ searches for the cells marked with $id(w)$s and collects set of $id'(f)$s from these cells. If any of the $id'(f)$ is in the deleted list, eliminate that $id'(f)$ from the collection and eliminate the cell with that $id'(f)$ and relink the cells accordingly.

$\mathcal{S}$ returns, $\widetilde{t_s} = (\gamma(id(w)), \widetilde{T_c}[\gamma(id(w))] \oplus \alpha_0, K_{id(w)})$, where $\gamma : \widetilde{T} \rightarrow \widetilde{T_c}$ is a PRP.

- *Simulating add token* t_a: Given $\mathcal{L}_{add}(f, \mathbf{f})$

1. If $\widetilde{T'}[id'(f)] \neq \perp$, then the file is already added and not deleted. In this case $\mathcal{S}$ returns the previous token t_a.
2. If $\widetilde{T'}[id'(f)] = \perp$, $\mathcal{S}$ adds the file and simulates the internal data structures as follows.
   (a) $\forall i \in [|\hat{f}|]$, ($w_i$'s are chosen according to the order in the leakage), $\mathcal{S}$ chooses a cell from the free list L_{free} ($\mathcal{L}$ gets it from $\mathcal{L}_{bld}(\mathbf{f})$) in $\widetilde{A}$ at the location α and does the followings.
      i. $\alpha_0 \leftarrow \widetilde{T}[id(w_i)]$
      ii. $\alpha_1 \leftarrow \widetilde{A}[\alpha_0][1]$
      iii. $\widetilde{A}[\alpha].[0] \leftarrow \alpha_0$; $\widetilde{A}[\alpha][1] \leftarrow \alpha_1$
      iv. If $\alpha_1 \neq \perp$, $\widetilde{A}[\alpha_1][0] \leftarrow \alpha$
      v. $\widetilde{A}[\alpha_1][1] \leftarrow \alpha$.
      vi. Mark $\widetilde{A}[\alpha]$ with $id'(f)$.
      vii. If $i = 1$, store α in $\widetilde{T'}[id'(f)]$ and α in α'
      else $\widetilde{A}[\widetilde{T}[id(w_{i-1})]][2] \leftarrow \alpha$
      If $i = |\hat{f}|$, set $\widetilde{A}[\widetilde{T}[id(w_i)]][2] \leftarrow \perp$

(b) $\forall i \in [|\hat{f}|]$,

i. Let $p_i = (\gamma(id(w_i)), \widetilde{T}[\gamma(id(w_i))], r_i^1, r_i^2, r_i^3, r_i^4, r_i^5, r_i)$, where r_i^1, r_i^2, r_i^3 and r_i^4 are random bit-strings of length $\log |A|$, r_i^5 is a random bit-string of length $\log l_{max}$, r_i is a random bit-strings of length λ.

ii. $\widetilde{A}'[\beta_i] \leftarrow (L \oplus r_i^1, R \oplus r_i^2, D \oplus r_i^3, Rs \oplus r_i^4, id'(f) \oplus r_i^5, r_i)$ where β_i is the index to the cell in $\widetilde{A}$ which was chosen while w_i added, $L = \widetilde{A}[\beta_i][0]$, $R = \widetilde{A}[\beta_i][1]$, $D = \widetilde{A}[\beta_i][2]$ and $Rs = \widetilde{A}[\beta_i][3]$

(c) $\mathcal{S}$ returns $\widetilde{t_a} = (\gamma'(id'(f)), \widetilde{T}'[\gamma'(id'(f))] \oplus \alpha', p_1, p_2, \ldots, p_{|\hat{f}|})$, the add token, where $\gamma : \widetilde{T}' \rightarrow \widetilde{T}'_c$ is a PRP.

- *Simulating delete token* t_d: Given $\mathcal{L}_{del}(f, \mathbf{f})$

1. If the file with $id'(f)$ is not previously added.

   (a) For the all previous searched queries $id(w)$, $\mathcal{S}$ checks whether the entries corresponding to $id(w)$ contains $id'(f)$. If contains, for some $id(w)$ in cell $\widetilde{A}[N]$, $\mathcal{S}$ adjusts the links of the right and left cell appropriately from $\widetilde{A}$ to delete $\widetilde{A}[N]$, adds $\widetilde{A}[N]$ to $\widetilde{L}_{free}$ and increases m_f counter by 1, where initially $m_f = 0$.

   (b) $\mathcal{S}$ selects $(|\hat{f}| - m_f)$ number of random unused cells from $\widetilde{A}$ and adds them to $\widetilde{L}_{free}$ according to $L_{free_{del}}$.

2. If the file with $id'(f)$ was previously added, then all cell with $id'(f)$ can be found by down links. So, $\mathcal{S}$ adds the cells from $\widetilde{A}$ and the corresponding nodes from $\widetilde{A}'$ to their free lists.

3. Return token $\widetilde{t_d} = (id'(f), \widetilde{T}'[id'(f)], K_{id'(f)})$

- *Simulating random oracle queries*

1. (Answering H_1, H_2 and H_3 queries) Given query $H_i(K, r)$, $i = 1, 2, 3$, $\mathcal{S}$ checks whether K is associated with some keyword identifier $id(w)$.

   (a) If not, it returns a random bit-string v of length $\log |A|$ and sets $RO_i[(K, r)] \leftarrow v$ to maintain consistency with future queries.

   (b) If so, $\mathcal{S}$ finds all entries in $\widetilde{A}$ marked with $id(w)$ and checks whether any of them has randomness r. If no such cell in $\widetilde{A}$ exists, return v as above. If such a cell in $\widetilde{A}$ exists, $\mathcal{S}$ returns $v \leftarrow RO_i[(K, r)]$

2. (Answering H_4 and H_5 queries) Given query (K, r), $\mathcal{S}$ checks whether K is associated with some keyword (file) identifier $id(w)$ ($id'(f)$).

(a) If not, it returns a random bit-string v' of length $\log|A|$ ($\log|l_{max}|$) and sets $RO_4[(K,r)] \leftarrow v'$ ($RO_5[(K,r)] \leftarrow v'$) to stay consistence with future queries.

(b) If yes, $\mathcal{S}$ finds all entries in $\widetilde{A}$ marked with $id(w)$ ($id'(f)$) and checks whether any of them in $\widetilde{A}$ has randomness r. If no such cell in $\widetilde{A}$ exists, return v' as above. If such a cell in $\widetilde{A}$ exists, $\mathcal{S}$ returns $v' \leftarrow RO_4[(K,r)]$ ($v' \leftarrow RO_5[(K,r)]$)

Now, since A and $\widetilde{A}$ are indistinguishable as they are distributed identically, indistinguishability of T (T') and $\widetilde{T}$ ($\widetilde{T'}$) follows from the pseudo-randomness of G (G'), $\mathbf{c}$ and $\widetilde{\mathbf{c}}$ are indistinguishable as SE is CPA secure, indistinguishability of t_s, t_a, t_d with respect to $\widetilde{t_s}$, $\widetilde{t_a}$, $\widetilde{t_d}$ follows from the pseudo-randomness of F, F', G, G' and P.

□

4.5 Performance evaluation

To demonstrate the performance of our algorithm, we implemented a prototype of our DSSE scheme Trids. The main code for Trids was written with C. All libraries were open source. For data pre-processing, we have used both Python and C. For file encryption, we have used AES where SHA-256 is taken as a hash function. We used a desktop with 8 GB DDR3 RAM, Intel Xeon E3-1200 Core i7-4770, and 1TB HDD.

To show the correctness and scalability of our algorithm we have compared our scheme with [48]. We note that they implemented with Intel Xeon CPU 2.26 GHz (L5520) running Windows Server 2008 R2. We have taken Enron e-mail dataset [33] to run our experiment. This dataset is publicly available. The downloaded data is 9.3 GB in pst format. The size of the data after extraction is 10.93 GB. The dataset consists of different categories like sent items, inbox, etc. We have considered only the content of the emails of each category for our experiments.

Experiment: The main functions of our experiments are written in C. In the pre-processing step, we have extracted the content of each email. Each of these extracted emails has a set of distinct keywords. We have considered the union of all these sets as the set of keywords W. We have tested our algorithms with random files taken from the Enron dataset. We have considered 6000 to 25000 random files (incrementing 1000 files each time). Every experiment is tested 100 times.

Time taken per keyword-file pair: Since index size is different as the different number of files has been added, the time taken by the total queries is different and proportional to the number of

files. However, we have observed that *the time taken is constant with respect to per keyword-file pair*. In Table 4.4, we have shown time taken per pair.

Table 4.4: Time Taken per keyword-file pair

Functions	$Build$	$AddTkn$	Add	$SearchTkn$	$Search$	$DeletTkn$	$Delete$
Time (μs/pair)	7.9368	7.1995	0.1190	0.0139	1.2162	0.1050	3.4868
Std. deviation	0.2852	0.1875	0.0017	0.0021	0.0028	0.0134	0.0427

According to the results shown in Table 4.4, the most expensive operation in the algorithms is hash calculation. Other operations have a very negligible effect as the standard deviation is very low. Except $Build()$ function, only $Add()$ has maximum execution time. This is because it has the maximum number of hash computation which is $5|\hat{f}|$, for a file f. On the other hand, the search token has taken minimum time as it has only 3 hash computation which is constant. Thus, it can be seen that if the execution time for any operation increases as soon as the number of files or database size increases in the system.

Time taken per query: We have analyzed the time taken per query. The time taken to generate tokens is independent of the size of the index and it is a client-side process. $AddToken()$ takes approximately 625 μs per query whereas $SearchToken()$ and $DeleteToken()$ take 2.68 μs and 5.88 μs per query respectively. This is because each of the files has 106 distinct keywords on an average. So, each AddToken query computes approximately 1000 hashes where $SearchToken()$ or $DeleteToken()$ has to compute only 3 hashes each. The average time to add a file to the index is 10.33 μs which is constant with respect to the size of the index but proportional to the number of distinct keywords contained in the file.

Table 4.5: Time Taken by per query

Functions	$AddTkn$	Add	$SearchTkn$	$DeletTkn$	$Delete$
Time (μs/query)	625.0	10.33	2.685	5.876	197.9
Std. deviation	16.28	0.146	0.013	0.117	24.53

Similarly, deletion time is proportional to the number of distinct keywords corresponding to the deleted file. Each deletion takes 369 μs on an average where the average number of distinct keywords in a file is 106. The deletion has to compute hashes thrice the number of distinct keywords which is 318 on an average. Table 4.5 shows the time taken by each query.

In our case, $SearchToken()$ and $DeleteToken()$ has the same amount of computation, but the search token takes less time because we have used a dictionary for the function F.

Only search time varies with the index size. Whenever the number of files increases, the number of keyword-file pairs increases. Thus, for the same query, the number of files increases in search query results. As we see in Fig. 4-7, the search time grows at a constant rate which is $1.2 \times 10^{-4} \mu s$/pair.

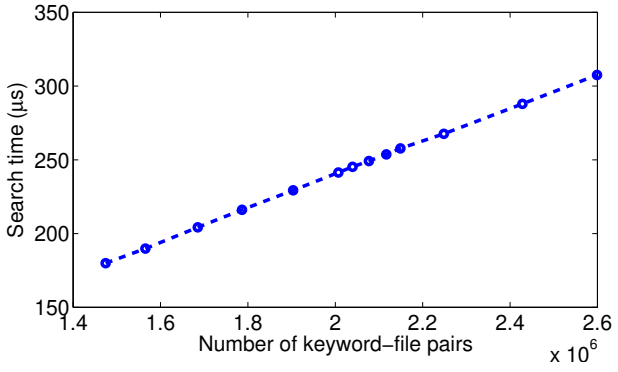


Figure 4-7: Number of keyword-file pairs vs. Search query time per search query

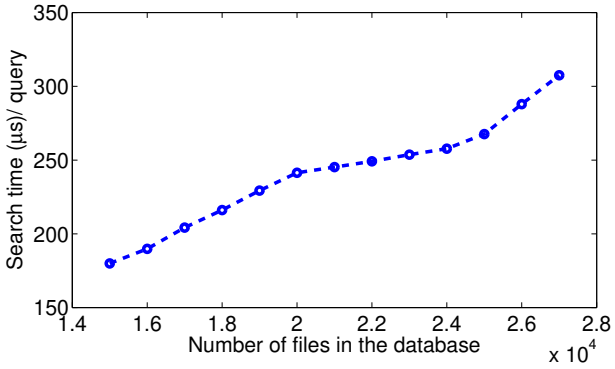


Figure 4-8: Number of files vs. Search query time per search query

Moreover, we have taken the number of added files into consideration. Fig. 4-8 shows that search time increases at an almost constant rate with the number of files in the database. The rate is $1.15 \times 10^{-2} \mu s$ per file. Thus, adding a file in the database increases each search query time by $1.15 \times 10^{-2} \mu s$.

Build time: Initial building time depends on the size of the database. We have analyzed $Build$ time with the database size. The database size is measured with respect to the number of keyword-file pairs, the number of files, and the size of the files.

From Fig. 4-9, we observe that $Build$ time increases proportionally with the number of keyword-file pairs in the database, though this time is almost constant with respect to each keyword-file pair. $Build$ time changes with respect to varying file-sizes (in MB) are shown in Fig. 4-10. Fig. 4-11 shows variations in $Build$ time with variable number of files. The line of regression is shown in the figure that indicates that the points are very close to the line.

4.5.1 Comparison with dynamic keyword search scheme by Kamara et al. [48]

In our test dataset, we only have considered the contents of the emails except for headers, footers, etc. We have compared the results of our scheme with those in the DSSE scheme proposed by Kamara et al. [48] (here, we call this scheme "KPR"). We have considered [48] for comparison as

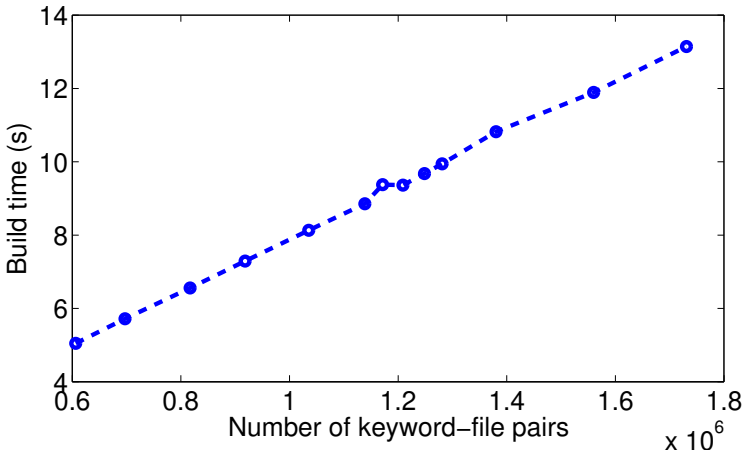


Figure 4-9: Number of pairs vs. Build time

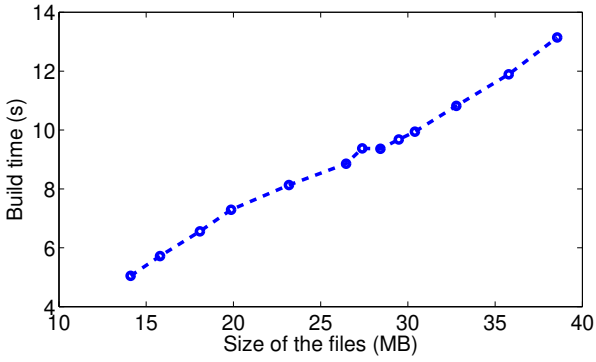


Figure 4-10: Size of the files vs. Build time

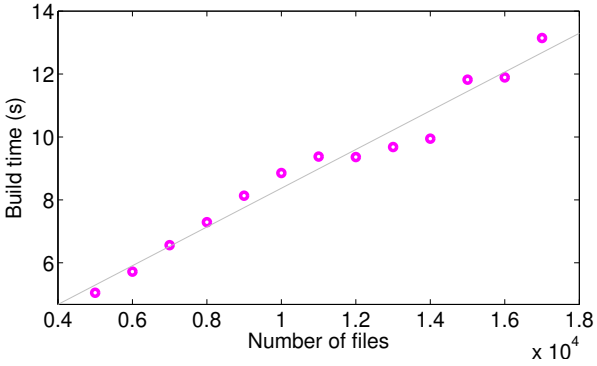


Figure 4-11: Number of files vs. Build time

it the closest to our scheme. We have taken the same size of the dataset, i.e., 4MB, 11 MB, and 16MB. We have compared the client-side $Build()$ function and the server-side $Add()$, $Search()$ and $Delete()$ functions. Since a client does not send a large number search or delete queries at once and token generation takes constant time for any size of database or files, we have ignored time to generate search tokens or delete tokens.

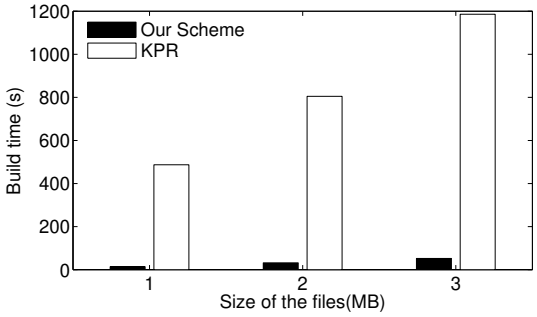


Figure 4-12: Build time comparison between KPR[48] and our scheme

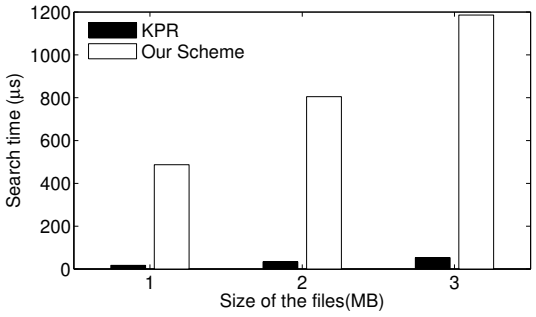


Figure 4-13: Search time comparison between KPR and our scheme

In Fig. 4-12, we have compared the $Build$ time in our scheme with that in [48]. We see that the building time grows very slowly in our scheme. However,in Fig. 4-13, while comparing search time, [48] gives different results. This may be due to implementing differently. With respect to computation, both the schemes have the same number of hash computations which is the only expensive operation.

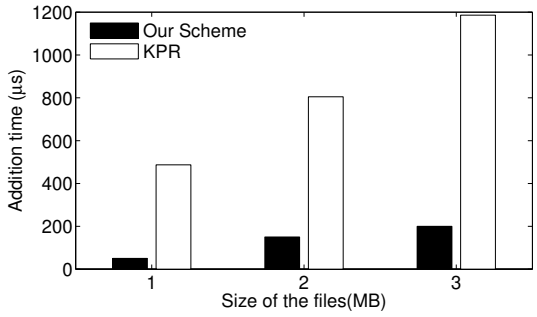


Figure 4-14: Add time comparison between KPR and our scheme

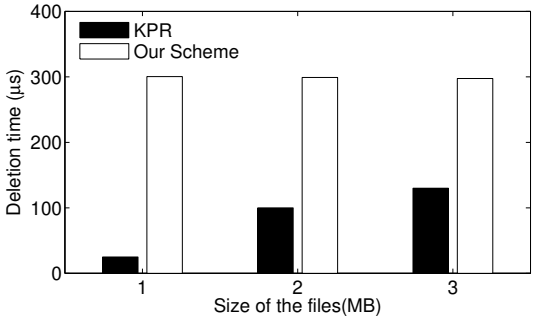


Figure 4-15: Delete time comparison between KPR and our scheme

$Add()$ time and $Delete()$ time are shown in the Fig. 4-14 and Fig. 4-15 respectively. To add a file into the database, the cloud server needs to update the links of the nodes equivalent to the size of the distinct keyword sets of that file. So, it proportional to the size of the files. Again $Add()$ does not have any hash computations which makes the taken time lower. $Delete()$ has hash computation

thrice the number of the distinct keyword sets of that file. However, this gives a higher level of privacy.

Conclusion

In this chapter, we have proposed an efficient DSSE scheme that has optimal search time with less deletion leakage compared to other DSSE schemes. In the next chapter, we study the DSSE schemes in the presence of malicious cloud servers. We propose a single keyword search DSSE scheme that is verifiable.

Chapter 5

A Forward private Publicly Verifiable Dynamic SSE scheme

As we have seen in Chapter 4, searchable symmetric encryption (SSE) scheme enables a client or data owner to store its data in a cloud server without losing the ability to search over them. Most of the SSE schemes like [32], [86], [26], [93], [27], etc. and the DSSE schemes like [87], [42], [21], [35], [47], [100], [79], [60], etc. considers the cloud server to be honest-but-curious. An honest-but-curious cloud server follows the protocol but wants to extract information about the plaintext data and the queries. However, if the cloud itself is malicious, it does not follow the protocol correctly. In the context of search, it can return only a subset of results, instead of all the records of the search. So, there is a need to verify the results returned by the cloud to the user. An SSE scheme for static data where the query results are verifiable is called Verifiable SSE (VSSE). Similarly, if the data is dynamic the scheme is said to be a verifiable dynamic SSE (VDSSE).

There are single keyword search VSSE schemes that are either new constructions supporting verifiability or design techniques to achieve verifiability on the existing SSE schemes by proposing generic algorithms. VSSE with single keyword search has been studied in [24], [28], [63]. In [91], [96] etc., VSSE scheme with conjunctive query has been studied. Moreover, there are also works that construct VDSSE schemes for both single keyword search ([66]) as well as complex query search including fuzzy keyword search ([112]) and Boolean query ([44]). However, most of them are *privately verifiable*. A VSSE or VDSSE scheme is said to be privately verifiable if the only user, who receives the search result, can verify it. On the other hand, a VSSE or VDSSE scheme is said to be *publicly verifiable* if any third party, including the database owner, can verify the search result without knowing its content.

There is also literature on public verifiability. Soleimanian and Khazaei [85] and Zhang et al. [109] have presented SSE schemes that are publicly verifiable. VSSE with Boolean range queries has been studied by Xu et al. [102]. Though their verification method is public, since the verification is based on blockchain databases, it has an extra monetary cost. Besides, Monir Azraoui [6] presented a conjunctive search scheme that is publicly verifiable. In the case of dynamic database, the publicly verifiable scheme by Jiang et al. [44] supports Boolean query, whereas Miao et al. [66] supports single keyword search.

However, file-injection attack [110], in which the client encrypts and stores files sent by the cloud server, recovers keywords from future queries, has forced researchers to think about dynamic SSE schemes to be forward private where adding a keyword-document pair does not reveal any information about the previous search result with that keyword. We have already discussed file injection attacks in Section 3.1.3 of Chapter 3.

Besides, in presence of a malicious cloud server, the owner can outsource the verifiability to a third-party auditor to reduce its computational overhead. The only forward private single keyword search VSSE scheme is proposed by Yoneyama and Kimura [107]. However, the scheme is privately verifiable and the owner requires a significant amount of computation for verification.

5.0.1 Our contribution

In this chapter, we have contributed the following in the literature of VSSE.

1. We have formally defined a verifiable DSSE scheme. Then we have proposed a generic publicly verifiable SSE scheme Aris which is very efficient and easy to integrate.
2. We have proposed a generic publicly verifiable dynamic SSE scheme Srica. Our proposed scheme is forward private. This property is necessary to protect a DSSE scheme from file injection attacks. However, no previous publicly verifiable scheme is forward private. In fact, only forward private scheme [107] is privately verifiable.
3. We have presented formal security proofs for these schemes and show that they are adaptively secure in the random oracle model.

On the owner side, both of the schemes do not use any more storage more than the underlying schemes from which they are derived. Thus, for a resource-constrained client, the schemes are very effective and efficient.

In Table 5.1, we have compared our proposed schemes with existing ones.

Table 5.1: Different verifiable SSE schemes

Data Type	static				dynamic			
Query Type	single		complex		single		complex	
Verification	private	public	private	public	private	public	private	public
Schemes	[24], [28], [74], [63], Aris	[85]	[96], [57], [102]	[85]	[107], [17]	[66], Srica	[112]	[44]
Forward private	not applicable				[107], Srica			

5.0.2 Organization

We have discussed the required preliminary topics in Section 5.1. In Section 5.2, we have presented a generic approach of verifiable SSE scheme. In Section 5.3, we have present our proposed generic construction of publicly verifiable DSSE scheme in details. We have compared its complexity with similar publicly verifiable schemes in Section 5.4.

5.1 Preliminaries

5.1.1 System model

In this section, we briefly describe the system model considered in this chapter. In our model of verifiable SSE, there are three entities–Owner, Auditor and Cloud. The system model is shown in the Fig. 5-1. We briefly describe them as follows.

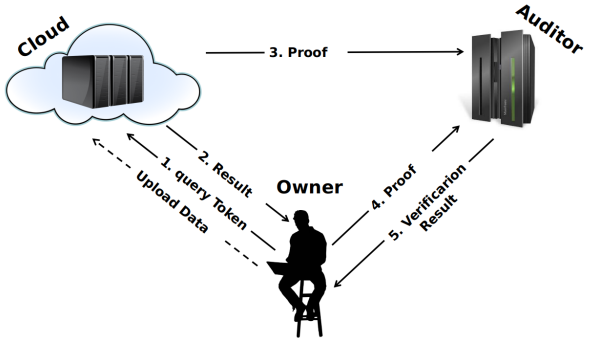


Figure 5-1: The system model of the a verifiable dynamic SSE scheme

1. **Owner:** Owner is the owner as well as user of the database. It is considered to be *trusted*. It builds an secure index, encrypts the data and then outsources both to the cloud. Later, it sends encrypted query to the cloud for searching. Therefore, it is an user as well. It is the client who requires the service.

2. **Cloud:** Cloud or the cloud server is the storage and computation service provider. It stores the encrypted data sent from the owner and gives result of the query requested by it. The cloud is assumed to be *malicious*. It can deviate from protocol by not only computing on, or not storing the data but also making the user fool by returning incorrect result.

3. **Auditor:** Auditor is an *honest-but-curious* authority which does not collude with the cloud. Its main role is to verify whether the cloud executes the protocol honestly. It tells the user whether the returned result is correct or not.

5.1.2 Design goals

Assuming the above system model, we aim to provide a solution to the verifiability problem of existing forward private schemes. In our design, we take care to achieve confidentiality, scalability, efficiency, and update support over the encrypted outsourced data, as described in Section 1.1.2.

Moreover, since it is observed previously that a DSSE scheme without forward privacy is vulnerable to even an honest-but-curious adversary. So, our target is to make a publicly verifiable DSSE scheme without losing its forward privacy property.

5.1.3 Definitions

$\mathcal{D}$ be the space of document identifiers and $\mathcal{DB}$ be the set of documents to be outsourced. Thus, $\mathcal{DB} \subseteq \mathcal{D}$. For each keyword $w \in \mathcal{W}$, the set of document identifiers that includes w is denoted by $DB(w) = \{id_1^w, id_1^w, \ldots, id_{c_w}^w\}$, where $c_w = |DB(w)|$ and $id_i^w \in \mathcal{DB}$. Thus, $\bigcup_{w \in \mathcal{W}} DB(w) \subseteq \mathcal{DB}$. Let $\overline{DB} = \{c_{id} : id \in \mathcal{D}\}$ where c_{id} denotes the encrypted document that has identifier id.

We assume that there is a one-way function H' that maps each identifier id to certain random numbers. These random numbers are used as document names corresponding to the identifier. The function is can be computed by both the owner and cloud. However, from a document name, the identifier cannot be recovered. Throughout, we use identifiers. However, when we say cloud returns documents to the owner, we assume the cloud performs the function on every file identifier before returning them.

Let, $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a cryptographic hash function, $\mathcal{H}$ be a bilinear hash, $R : \{0,1\}^* \rightarrow \{0,1\}^*$ be a PRNG and $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a HMAC.

5.1.4 Verifiable Dynamic Searchable Symmetric Encryption (VDSSE)

An SSE scheme allows a client to outsource a dataset it owns to a cloud service provider in encrypted form without losing the ability to perform queries over the data. The most popular query is the keyword search where the dataset is a collection of documents. The client can retrieve partially

encrypted data without revealing any meaningful information to the cloud. Throughout we take a query as a single keyword search query.

A *dynamic SSE* (DSSE) scheme is an SSE scheme that supports updates. A *Verifiable DSSE* (VDSSE) scheme is a DSSE scheme together with verifiability. The verification can be done either by an external auditor or the owner. The primary reason to bring an auditor is to reduce computational costs of verifiability at the owner-side. This allows an owner to be lightweight.

Though a VDSSE scheme supports updates, we do not verify whether the cloud updates the database correctly or not. We only want to get the correct result with respect to the current state of the database. If the cloud updates the database incorrectly, it cannot give the actual result. Due to verifiability, it will be failed in the verification process to the auditor. We define a verifiable DSSE scheme formally as follows.

Definition 5.1 (Verifiable Dynamic SSE). *A verifiable dynamic SSE (VDSSE) scheme Ψ is a tuple* $(\texttt{VKeyGen}, \texttt{VBuild}, \texttt{VSearchToken}, \texttt{VSearch}, \texttt{VUpdateToken}, \texttt{VUpdate})$ *of algorithms defined as follows.*

- $K \leftarrow \texttt{VKeyGen}(1^\lambda)$*: It is a probabilistic polynomial-time (PPT) algorithm run by the owner. Given security parameter λ it outputs a key K.*
- $(\overline{DB}, \gamma) \leftarrow \texttt{VBuild}(K, \mathcal{DB})$*: The owner run this PPT algorithm. Given a key K and a set of documents $\mathcal{DB}$, it outputs the encrypted set of documents $\overline{DB}$ and an encrypted index γ.*
- $\tau_s \leftarrow \texttt{VSearchToken}(K, w)$*: On input a keyword w and the key K, the owner runs this PPT algorithm to output a search token τ_s.*
- $(R_w, \nu_w) \leftarrow \texttt{VSearch}(t_s, \gamma)$*: It is a PPT algorithm run by the cloud and the auditor collaboratively that returns a set of document identifiers result R_w to the owner with verification bit ν_w.*
- $\tau_u \leftarrow \texttt{VUpdateToken}(K, id)$*: It is a owner-side PPT algorithm that takes the key K and a document identifier id and outputs a update token τ_u.*
- $(\overline{DB}', \gamma') \leftarrow \texttt{VUpdate}(\tau_u, op, \gamma, \overline{DB})$*: It is a PPT algorithm run by the cloud. It takes an update token τ_u, operation bit op, the encrypted document set $\overline{DB}$ and the index γ and outputs updated $(\overline{DB}', \gamma')$.*

Computational Correctness A VDSSE scheme Ψ is said to be *correct* if $\forall \lambda \in \mathsf{N}$, $\forall K$ generated using $\texttt{KeyGen}(1^\lambda)$ and all sequences of search and update operations on γ, every search outputs the correct set of identifiers, except with a negligible probability.

Verifiability Note that, when we are saying a scheme is verifiable, it means that it verifies whether the search result is from the currently updated state of the database according to the owner. During an update query, it does not verifies whether the update is correctly done by the cloud or not. Instead, it verifies during the search and checks whether the search result is from the updated database. For example, if an owner added a document with some keywords and the cloud does not update the database. Later, if the owner searches with some keywords present in the document, and it should get the identifier of the document in the result set. Then, the result can be taken as verified.

5.1.5 Security definitions

We follow the security definition of [85]. There are two parts in the definition– confidentiality and soundness. We define security in an adaptive adversary model where the adversary can send queries depending on the previous results. Typically, most of the dynamic SSE schemes define their security in this model.

A DSSE, that does not consider verifiability, considers honest-but-curious (HbC) cloud server. In these cases, the owner of the database allows some leakage on every query made. However, it guarantees that no meaningful information about the database is revealed other than the allowed leakages. Soundness definition ensures that the results received from the cloud server are correct.

5.1.5.1 Confidentiality

Confidentiality ensures that a scheme does not give any meaningful information other than it is allowed. In our model, we have considered the cloud to be malicious. However, the auditor is HbC. Since verifiability has some monetary cost for the owner, it wants verifiability only when it is required. Also, the auditor does not have the database and ability to search. Given the proof, it only verifies the result. Thus, if the scheme is secure from the cloud, it is so from the auditor. Again, we have assumed that the cloud and the auditor do not collude. Hence, we do not consider the auditor in our definition of confidentiality.

Definition 5.2 (CKA2-Confidentiality of a verifiable DSSE scheme). *Let* $\Psi = (\texttt{VKeyGen}, \texttt{VBuild}, \texttt{VSearchToken}, \texttt{VSearch}, \texttt{VUpdateToken})$ *be a verifiable DSSE scheme. Let* $\mathcal{A}$*,* $\mathcal{C}$ *and* $\mathcal{S}$ *be a stateful adversary, a challenger and a stateful simulator respectively. Let* $\mathcal{L}$=$(\mathcal{L}_{bld}, \mathcal{L}_{srch}, \mathcal{L}_{updt})$ *be a stateful leakage algorithm. Let us consider the following two games.*

Real$_{\mathcal{A}}(\lambda)$*:*

1. *The challenger $\mathcal{C}$ generates a key $K \leftarrow \texttt{VKeyGen}(1^{\lambda})$.*
2. *$\mathcal{A}$ generates and sends $\mathcal{DB}$ to $\mathcal{C}$.*
3. *$\mathcal{C}$ builds $(\overline{DB}, \gamma) \leftarrow \texttt{VBuild}(K, \mathcal{DB})$ and sends $(\overline{DB}, \gamma)$ it to $\mathcal{A}$.*
4. *$\mathcal{A}$ makes a polynomial number of adaptive queries. In each of them, it sends either a search query for a keyword w or an update query for a keyword-document pair (w, id) and operation bit op to $\mathcal{C}$.*
5. *$\mathcal{C}$ returns either a search token $\tau_s \leftarrow \texttt{VSearchToken}(K, w)$ or an update token $\tau_u \leftarrow \texttt{VUpdateToken}(K, id)$ to $\mathcal{A}$ depending on the query.*
6. *Finally $\mathcal{A}$ returns a bit b that is output by the experiment.*

Ideal$_{\mathcal{A},\mathcal{S}}(\lambda)$*:*

1. *$\mathcal{A}$ generates a set $\mathcal{DB}$ of documents and gives it to $\mathcal{S}$ together with $\mathcal{L}_{bld}(\mathcal{DB})$.*
2. *$\mathcal{S}$ generates $(\overline{DB}, \gamma)$ and sends it to $\mathcal{A}$*
3. *$\mathcal{A}$ makes a polynomial number of adaptive queries q. For each query, $\mathcal{S}$ is given either $\mathcal{L}_{srch}(w, \mathcal{DB})$ or $\mathcal{L}_{updt}(op, w, id)$ depending on the query.*
4. *$\mathcal{S}$ returns, depending on the query q, to $\mathcal{A}$ either search token τ_s or update token τ_u.*
5. *Finally $\mathcal{A}$ returns a bit b' that is output by the experiment.*

We say Ψ is $\mathcal{L}$-secure against adaptive dynamic chosen-keyword attacks if $\forall$ PPT adversary $\mathcal{A}$, $\exists$ a simulator $\mathcal{S}$ such that

$$|Pr[\textbf{\textit{Real}}_{\mathcal{A}}(\lambda) = 1] - Pr[\textbf{\textit{Ideal}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mu(\lambda) \tag{5.1}$$

where $\mu(\lambda)$ is negligible in λ.

5.1.5.2 Soundness

The soundness property ensures that if a malicious cloud tries to fool the owner by returning incorrect result it will be caught to the auditor. We present game-based definition of soundness as follows.

Definition 5.3 (Soundness of a verifiable DSSE scheme). *Let Ψ be a verifiable DSSE scheme with $\Psi = (\texttt{VKeyGen}, \texttt{VBuild}, \texttt{VSearchToken}, \texttt{VSearch}, \texttt{VUpdateToken})$. Let us consider the following game.*

sound$_{\mathcal{A},\Psi}(\lambda)$*:*

1. *The challenger $\mathcal{C}$ generates a key $K \leftarrow \texttt{VKeyGen}(1^\lambda)$.*
2. *$\mathcal{A}$ generates and sends $\mathcal{DB}$ to $\mathcal{C}$.*
3. *$\mathcal{C}$ computes $(\overline{DB}, \gamma) \leftarrow \texttt{VBuild}(K, \mathcal{DB})$ and sends $(\overline{DB}, \gamma)$ to $\mathcal{A}$.*
4. *$\mathcal{A}$ makes a polynomial number of adaptive queries. In each of them, it sends either a search query for a keyword w or an update query for a keyword-document pair (w, id) and operation bit op to $\mathcal{C}$.*
5. *$\mathcal{C}$ returns either a search token $\tau_s \leftarrow \texttt{VSearchToken}(K, w)$ or an update token $\tau_u \leftarrow \texttt{VUpdateToken}(K, id)$ to $\mathcal{A}$ depending on the query.*
6. *After making polynomial number of queries, $\mathcal{A}$ chooses a target keyword w and send search query to $\mathcal{C}$.*
7. *$\mathcal{C}$ returns a search token τ_s. $\mathcal{A}$ executes and gets (R_w, ν_w) where $\nu_w = accept$ is verification bit from $\mathcal{C}$.*
8. *$\mathcal{A}$ generates pair (R_w^*) for a keyword w and gets verification bit $\nu_w^* = accept$.*
9. *If $\nu_w^* = accept$ even when $R_w^* \neq DB(w)$, $\mathcal{A}$ returns 1 as output of the game, otherwise returns 0.*

We say that Ψ is sound *if $\forall$ PPT adversaries $\mathcal{A}$, $Pr[$**sound**$_{\mathcal{A},\Psi}(\lambda) = 1] \leq \mu(\lambda)$.*

5.2 Verifiable SSE with static data

Since, in a verifiable SSE scheme, there is no update, it does not have `VUpdate` or `VUpdateToken` operation. We present a generic scheme that that converts an SSE scheme to a VSSE scheme. *Our target is to achieve verifiability, in presence of malicious cloud server, without loosing any other security property with minimal communication and computational costs.*

5.2.1 Issues with the existing verifiable SSE schemes

There are works that considered static SSE schemes and suggested authentication tag generation using MAC to protect the integrity of the search result. For each keyword w, they generate a tag $tag_w = H(id_1^w||id_2^w||\ldots||id_{c_w}^w)$ where H is a one-way hash function. Trivially, if the tags are stored at the owner's side then the scheme becomes privately verifiable. In that case, when a search is required, the owner can check integrity after receiving the result from the cloud.

However, this integrity checking does not protect the SSE scheme from malicious adversary *if the tags are outsourced* to the cloud. Checking integrity provides security only from honest-but-curious cloud servers. Let us consider an example. Suppose a keyword $w \in \mathcal{W}$ is searched and cloud gets the result $R_w = \{id_1^w, id_2^w, \ldots, id_{c_w}^w, tag_w\}$. Later, if some other keyword w' is searched, the cloud can return the same result and will pass the integrity checking.

5.2.2 A generic verifiable SSE scheme without client storage

Since it is desirable to outsource the data as well as tags to the cloud, the above result shows that checking integrity in the above way cannot be considered. It is easy to see that the scheme with checking the integrity of the result identifiers is not enough because there is no binding of the keyword with the tags. Here, we present a generic idea that makes any SSE scheme verifiable.

Scheme Description Let $\Sigma_s = (\texttt{KeyGen}, \texttt{Build}, \texttt{SearchToken}, \texttt{Search})$ be a result revealing static SSE scheme. We present a VSSE scheme Ψ_s=($\texttt{VKeyGen}$, $\texttt{VBuild}$, $\texttt{VSearchToken}$, $\texttt{VSearch}$) for static database as follows.

Let H be a one-way hash function and a key K' is chosen at random. For each keyword $w \in \mathcal{W}$, a key $k_w = H(K', w)$ is generated. k_w is then used to bind the keyword with corresponding tag $tag_w = H(k_w||id_1^w||id_2^w||\ldots||id_{c_w}^w)$. Finally, for each keyword w, $\{id_1^w, id_2^w, \ldots, id_{c_w}^w, tag_w\}$ is encrypted at build phase. Thus, while performing search with a keyword w, as search result, the

owner receives $\{id_1'^w, id_2'^w, \ldots, id_{c_w}'^w, tag_w'\}$. The owner accepts it if the regenerated tag tag_w' from the received identifiers is matched with the received one.

So, the main idea of the scheme is that instead of generating tags only with identifiers, they are bound with k_w which is dependent on w and can be computed by the owner only. After the search, if the cloud returns an incorrect set of document identifiers then the tag won't get matched. The scheme is shown in Fig. 5-2. We call the scheme Aris.

$\Psi_s.\texttt{VKeyGen}(1^\lambda)$

1. $K_{\Sigma_s} \leftarrow \Sigma_s.\texttt{KeyGen}(1^\lambda)$
2. $K' \xleftarrow{\$} \{0,1\}^\lambda$
3. Return $K_{\Psi_s} = (K', K_{\Sigma_s})$

$\Psi_s.\texttt{VBuild}(\mathcal{DB}, K_{\Psi_s})$

1. $(K', K_{\Sigma_s}) \leftarrow K_{\Psi_s}$
2. **for** each $w \in \mathcal{W}$
   (a) $k_w \leftarrow H(K'||w)$
   (b) $tag_w \leftarrow H(k_w||id_1^w||id_2^w||\ldots||id_{c_w}^w)$
   (c) $DB'(w) \leftarrow DB(w) \cup \{tag_w\}$
3. $\mathcal{DB}' \leftarrow \cup_{w\in\mathcal{W}} DB'(w)$
4. $(\gamma, \overline{DB}) \leftarrow \Sigma_s.\texttt{Build}(\mathcal{DB}', K_{\Sigma_s})$
5. Return $(\gamma, \overline{DB})$

$\Psi_s.\texttt{VSearchToken}(w, K_{\Sigma_s})$

1. $\tau_{\Sigma_s} \leftarrow \Sigma_s.\texttt{SearchToken}(w, K_{\Sigma_s})$
2. Return τ_{Σ_s}

$\Psi_s.\texttt{VSearch}(\gamma, \tau_{\Sigma_s})$

1. $(K', K_{\Sigma_s}) \leftarrow K_{\Psi_s}$
2. $\tau_{\Sigma_s} \leftarrow \Sigma_s.\texttt{SearchToken}(w, K_{\Sigma_s})$
3. $R_w \leftarrow \Sigma_s.\texttt{Search}(\gamma, \tau_{\Sigma_s})$
4. $k_w \leftarrow H(K'||w)$
5. $\{id_1'^w, id_2'^w, \ldots, id_{c_w}'^w, tag_w'\} \leftarrow R_w$
6. $tag_w \leftarrow H(k_w||id_1'^w||id_2'^w||\ldots||id_{c_w}'^w)$
7. Accept R_w if $tag_w' = tag_w$

Figure 5-2: Algorithm for generic verifiable SSE scheme Aris

Note that, for the static case, tag computation is enough to validate a result. Since one-way hash computation is very efficient and requires a small amount of resources, we do not consider any external authority like an auditor for verifiability. So, the scheme is privately verifiable.

Cost for verifiability The cloud storage is increased by $|\mathcal{W}|$ tags. However, depending on the scheme the actual increment might be less than $|\mathcal{W}|$ tags but still it is asymptotically $O(|\mathcal{W}|)$. The communication cost for verification is only increased by one tag from the cloud to the owner. If we consider computation, to verify a search result, the owner only has to compute a hash value which is very little.

Soundness In case the cloud does not want to perform a search properly, then it cannot get the identifiers and the corresponding tag. So, it has to send either random identifiers or identifiers corresponding to some other searched keyword. In both cases, It cannot be passed verifiability test to the owner.

Confidentiality The confidentiality of our proposed scheme follows from the security of the embedded SSE scheme.

5.3 Our proposed Forward private Publicly Verifiable DSSE scheme

In this section, we propose a simple generic dynamic SSE scheme Srica which is forward private as well as verifiable. Let Σ_f = (`KeyGen`, `Build`, `Search`, `SearchToken`, `Update`, `UpdateToken`) be a result revealing forward private dynamic SSE scheme.

It is to be noted that any forward private SSE scheme stores the present state of the database on the client-side. Corresponding to each keyword, most of them stores the number of documents containing it. Let $C = \{c_w : w \in \mathcal{W}\}$ be the list of such numbers.

Since it considers any forward private scheme Σ_f, it only adds an additional encrypted data structure to make the scheme verifiable. The algorithms of our proposed scheme Srica, denoted by Ψ_f, are given in Figure 5-3. They are divided into three phases– initialization, search and update.

Initialization phase: In this phase, secret and public keys are generated by the owner, and thereafter the encrypted searchable structure is built. During key generation, three types of keys are generated– K_{Σ_f} for the Σ_f; (sk, pk) for the bilinear signature scheme; and two random strings K_s, K_t for seed and tag generation respectively.

Thereafter, a signature table T_{sig} is generated, before building the secure index γ and encrypted database $\overline{DB}$, to store the signature corresponding to each keyword-document pair. For each pair (w, id_i^w), the position $pos_i^w = F(tag_w, id_i^w||i)$ is generated with a HMAC F. The position is actually act as key of a key-value pair for a dictionary. The document identifier is bounded with pos_i^w together with $tag_w = F(K_t, w)$. The tag_w is fixed for a keyword and is given to the cloud server to find pos_i^w. The signature σ_i^w for the same pair is also bounded with random number r_i^w which can only be generated from PRG R with the seed s_w. Then (σ_i^w, pos_i^w) pair is added in the

$\Psi_f.\texttt{VKeyGen}(1^\lambda)$

1. $K_{\Sigma_f} \leftarrow \Sigma_f.\texttt{KeyGen}(1^\lambda)$
2. $(sk, pk) \leftarrow \mathcal{S}.\texttt{Setup}(1^\lambda)$
3. $K_s \leftarrow \{0,1\}^\lambda$
4. $K_t \leftarrow \{0,1\}^\lambda$
5. Return $K_{\Psi_f} = (K_t, K_s, sk, pk, K_{\Sigma_f})$

$\Psi_f.\texttt{VBuild}(\mathcal{DB}, K_{\Psi_f})$

1. $T_{sig} \leftarrow$ empty list of size $|\mathcal{W}|$
2. **for** $w \in \mathcal{W}$
   (a) $s_w \leftarrow F(K_s, w)$; $tag_w \leftarrow F(K_t, w)$
   (b) **for** $i = 1$ **to** $c_w (= |DB(w)|)$
      i. $r_i^w \leftarrow R(s_w||i)$;
      ii. $m_i^w \leftarrow r_i^w.id_i^w \mod q$
      iii. $\sigma_i^w \leftarrow \mathcal{S}.\texttt{Sign}(sk, m_i^w)$
      iv. $pos_i^w \leftarrow F(tag_w, id_i^w||i)$
      v. $T_{sig}[pos_i^w] \leftarrow \sigma_i^w$
3. $(\gamma, \overline{DB}) \leftarrow \Sigma_f.Build(\mathcal{DB}, K_{\Sigma_f})$
4. Return $(\gamma, \overline{DB}, T_{sig})$ to the cloud

$\Psi_f.\texttt{VSearchToken}(w, K_{\Psi_f})$

1. $(K_t, K_s, sk, pk, K_{\Sigma_f}) \leftarrow K_{\Psi_f}$
2. $\tau_{\Sigma_f} \leftarrow \Sigma_f.\texttt{SearchToken}(w, K_{\Sigma_f})$
3. $tag_w \leftarrow F(K_t, w)$;
4. $\tau_s^{\Psi_f} \leftarrow (\tau_{\Sigma_f}, tag_w)$
5. Return $\tau_s^{\Psi_f}$ to cloud

$\Psi_f.\texttt{VSearch}(\gamma, \tau_s^{\Psi_f})$

Cloud:

1. Receive $\tau_{\Psi_f} = (\tau_{\Sigma_f}, tag_w)$ from Owner
2. $\{id_1'^w, \ldots, id_{c_w'}'^w\} \leftarrow \Sigma_f.\texttt{Search}(\gamma, \tau_{\Sigma_f})$
3. **for** $i = 1$ **to** c_w'
   (a) $pos_i^w \leftarrow F(tag_w, id_i'^w||i)$
   (b) $\sigma_i' \leftarrow T_{sig}[pos_i^w]$;
4. $\sigma' \leftarrow \prod_{i=1}^{c_w'} \sigma_i'$
5. $R_w \leftarrow \{id_1'^w, id_2'^w, \ldots, id_{c_w'}'^w\}$
6. $pf_c \leftarrow \sigma'$
7. Return pf_c to auditor and R_w to Owner

Owner:

1. Receives R_w
2. $c_w \leftarrow C[w]$
3. If $c_w \neq c_w'$ **Return** reject bit.
4. $s_w \leftarrow F(K_s, w)$
5. **for** $i = 1$ **to** c_w **do**
   (a) $r_i^w \leftarrow R(s_w||i)$
   (b) $m_i^w \leftarrow id_i'^w.r_i^w \mod q$
6. $m = \sum_{i=1}^{c_w} m_i^w \mod q$
7. Send $pf_o = m$ to the auditor

Auditor:

1. Receives $pf_o = m$ from owner and $pf_c = \sigma'$ from cloud
2. $b_v \leftarrow \mathcal{S}.\texttt{Verify}(pk, m, \sigma')$
3. If $b_v = failure$, **Return** *reject*

$\Psi_f.\texttt{VUpdateToken}(K_{\Psi_f}, w, id)$

1. $\tau_u \leftarrow \Sigma_f.\texttt{UpdateToken}(K_{\Sigma_f}, w, id)$
2. Return τ_u

$\Psi_f.\texttt{VUpdate}(T_{tag}, \gamma, \tau_u)$

Owner:

1. $\{w_1, w_2, \ldots, w_{n_{id}}\} \in id$
2. **for** $i = 1$ **to** n_{id}
   (a) $\tau_u \leftarrow \Psi_f.\texttt{VUpdateToken}(K_{\Psi_f}, w_i, id)$ $\forall i \in [c_w]$
   (b) $b_v \leftarrow \Sigma_f.\texttt{Update}(\gamma, \tau_u)$
   (c) **if** $b_v \neq succsess$ Return
3. **for** $i = 1$ **to** n_{id}
   (a) $tag_{w_i} \leftarrow F(K_t, w_i)$
   (b) $c_{w_i} \leftarrow C[w_i]$
   (c) $s_w \leftarrow F(K_s, w)$;
   (d) $r \leftarrow R(s_w||(c_{w_i} + 1))$
   (e) $m \leftarrow id.r \mod q$
   (f) $\sigma_i \leftarrow \mathcal{S}.\texttt{Sign}(sk, m)$
   (g) $pos_i \leftarrow F(tag_{w_i}, id||(c_{w_i} + 1))$
   (h) $C[w] = C[w] + 1$
4. $pos \leftarrow \{pos_1, pos_2, \ldots, pos_{n_{id}}\}$
5. $\sigma \leftarrow \{\sigma_1, \sigma_2, \ldots, \sigma_{n_{id}}\}$
6. send $\tau_u^{\Psi_f} = (pos, \sigma)$ to cloud

Cloud:

1. $\{pos_1, pos_2, \ldots, pos_{n_{id}}\} \leftarrow pos$
2. $\{\sigma_1, \sigma_2, \ldots, \sigma_{n_{id}}\} \leftarrow \sigma$
3. $T_{sig}[pos_i] \leftarrow \sigma_i$, $\forall i \in [n_{id}]$

Figure 5-3: Generic verifiable dynamic SSE scheme Srica without extra client storage

table T_{sig} as key-value pair. After the building process, the owner outsources γ, $\overline{DB}$ and T_{sig} to the cloud.

Search Phase: In this phase, the owner first generates a search token τ_{Σ_f} to search on Σ_f. Then, it regenerates tag_w and the seed s_w and then, sends them to the cloud.

The cloud performs search operation according to Σ_f and use the result identifiers $\{id_1, id_2, \ldots id_{c'_w}\}$ to gets the position in T_{sig} corresponding to each pair. It is not able to generate the positions if it does not search for the document identifiers. It collects the signatures stored in those positions, multiplies them, and sends multiplication results to the auditor as its part pf_c of the proof. It sends the search result to the owner.

The owner first generates random numbers $\{r_1, r_2, \ldots r_{c'_w}\}$ and regenerates aggregate message $m = \sum_{i=1}^{i=c'_w} r_i.id_i^w \pmod q$ of the identifiers and sends m to the auditor as pf_o, owner's part of the proof. After receiving pf_c and pf_o, the auditor only computes $\mathcal{S}.\texttt{Verify}(pk, m, \sigma')$. It outputs accept if signature verification returns *success*. We can see that no information about the search results is leaked to the auditor during verification.

Update Phase: In our scheme, while adding a document, instead of being updated only a keyword-document pair, we assume that all such pairs corresponding to the document are added. To add a document with identifier id and keyword set $\{w_1, w_2, \ldots, w_{n_{id}}\}$, the owner generates the position and the corresponding signature for each containing keyword. The cloud gets them from the owner and adds them in the table T_{sig}.

Correctness For correctness it is enough to check the following.

$$\hat{e}(\mathcal{H}(m), pk) = \hat{e}(g^m, g^\alpha) = \hat{e}(g^{\alpha \sum m_i}, g) = \hat{e}(\prod g^{\alpha m_i}, g) = \hat{e}(\prod \sigma_i, g) = \hat{e}(\sigma, g)$$

Cost for verifiability We achieve, forward privacy as well as public verifiability without client storage in Ψ_f, our proposed scheme Srica. This increases the cloud storage by $O(N)$, where N is the number of document-keyword pairs. The proof has two parts one from the client and another from the owner. For a keyword, w, the sizes of them are one group element and one random λ-bit string only. Thus Auditor receives one element from both. The owner has to compute R_w integer multiplication and addition and then has to send one element.

Forward privacy We can see that while adding a document, it only adds some keyword-document pair, in the form of key-value pairs. So, During addition, the cloud server is adding key-value pairs in the dictionary. From these pairs, it cannot guess the keywords present in it. Again, when it

performs searches, it gets about the key (i.e., position on the table) only when it gets the identifiers. The one possibility to get the newly added key-value pair linked with the previous is if the added document gives the identifier of it. Since the one-way function, H' gives the document-name of the adding document, the cloud server cannot linked it with the previously searched keywords.

5.3.1 Security

The security of the scheme is shown in two parts– confidentiality and soundness.

Soundness The cloud server can cheat the owner in three ways by sending–

1. Incorrect number of identifiers– but it is not possible as the owner keeps the number of identifiers.
2. Same size result of other keywords– m is generated with a random number which can be generated only with the searched keyword and signatures are bound with that. So, the signature verification will be failed.
3. Result with some altered identifiers– since signatures are bounded with keywords and the random number, altering any will change m and similarly, the signature verification will be failed.

Thus the owner always will get the correct set of document identifiers.

5.3.1.1 Confidentiality

Let $\mathcal{L}^{\Sigma_f} = (\mathcal{L}^{\Sigma_f}_{bld}, \mathcal{L}^{\Sigma_f}_{srch}, \mathcal{L}^{\Sigma_f}_{updt})$ the leakage function of Σ_f. Let $\mathcal{L}^{\Psi_f} = (\mathcal{L}^{\Psi_f}_{bld}, \mathcal{L}^{\Psi_f}_{srch}, \mathcal{L}^{\Psi_f}_{updt})$ be the leakage function of Ψ_f, given as follows.

$$\begin{aligned}
\mathcal{L}^{\Psi_f}_{bld}(\mathcal{DB}) &= \{\mathcal{L}^{\Sigma_f}_{bld}(\mathcal{DB}), |T_{sig}|\} \\
\mathcal{L}^{\Psi_f}_{srch}(w) &= \{\mathcal{L}^{\Sigma_f}_{srch}(w), \{(id^w_i, pos^w_i, \sigma^w_i) : i = 1, 2, \ldots, c_w\}\} \\
\mathcal{L}^{\Psi_f}_{updt}(f) &= \{id, \{(\mathcal{L}^{\Sigma_f}_{updt}(w_i, id), pos^{w_i}, \sigma^{w_i}) : i = 1, 2, \ldots, n_{id}\}\}
\end{aligned}$$

We show that Ψ is $\mathcal{L}^{\Psi_f}$-secure against adaptive dynamic chosen-keyword attacks in the random oracle model, in the following theorem.

Theorem 5.1. *If F is a PRF, R is a PRG and Σ_f is $\mathcal{L}^{\Sigma_f}$-secure, then Ψ_f is $\mathcal{L}^{\Psi_f}$-secure against adaptive dynamic chosen-keyword attacks.*

Proof. To prove the above theorem, it is sufficient to show that there exists a simulator $\mathtt{Sim}_{\Sigma_f}$ such that $\forall$ PPT adversary $\mathcal{A}$, the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathtt{Sim}_{\Sigma_f}}(\lambda)$ are computationally indistinguishable.

We construct such a simulator $\mathtt{Sim}_{\Sigma_f}$ which adaptively simulates the extra data structure T_{sig} and query tokens. Let $\mathtt{Sim}_{\Sigma_f}$ be the simulator of the Σ_f. We simulate the algorithms as follows.

Simulating F**:** We simulate R with a table RO. Given (x, y), If $RO[(x, y)] = \perp$, then do $RO[(x, y)] \leftarrow \{0,1\}^\lambda$ and return $RO[(x, y)]$, else return the existing value $RO[(x, y)]$.

Simulating Build: Leakage function is given by $\mathcal{L}^{\Psi_f}_{bld}(\mathcal{DB}) = \{\mathcal{L}^{\Sigma_f}_{bld}(\mathcal{DB}), |T_{sig}|\}$. Let S_{bld} be returned by the simulator $\mathtt{Sim}_{\Sigma_f}$. Let us consider a table $\widetilde{T}_{tag}$. For each keyword w it stores a random λ-bit string. On input w, it returns $\widetilde{tag}_w \leftarrow \widetilde{T}_{tag}(w)$. $\mathtt{Sim}_{\Psi_f}$ keeps an extra table $\widetilde{T}'_{sig}$ such that it indicates whether the entry is queried or not. The simulation is done as follows.

1. Take empty tables $\widetilde{T}_{sig}$ and $\widetilde{T}'_{sig}$
2. For each $i = 1$ to $i = |T_{sig}|$ do
   (a) $pos_i \xleftarrow{\$} \{0,1\}^\lambda$; $r'_i \xleftarrow{\$} \{0,1\}^\lambda$
   (b) $val_i \xleftarrow{\$} g^{r'_i}$
   (c) $\widetilde{T}_{sig}[pos_i] \leftarrow val_i$
   (d) $\widetilde{T}'_{sig}[pos_i] \leftarrow 0$
3. Simulate Σ_f with $S_{bld} \leftarrow \mathtt{Sim}_{\Sigma_f}(\mathcal{DB})(\mathcal{L}^{\Sigma_f}_{bld}(\mathcal{DB}))$
4. return $(S_{bld}, \widetilde{T}_{sig})$ and keeps $\widetilde{T}'_{sig}$

Simulating Search token: Leakage function for a queried keyword w is given by $\mathcal{L}^{\Psi_f}_{srch}(w) = \{\mathcal{L}^{\Sigma_f}_{srch}(w), \{(id^w_i) : i = 1, 2, \ldots, c_w\}\}$.

We keep a table RO where $(\widetilde{tag}_w, id, i)$ is the key and pos is the value. Given search leakage corresponding to the keyword w, $\mathtt{Sim}_{\Psi_f}$ does the following things.

1. If $\widetilde{T}_{tag}[w]$ is null, i.e, the keyword is searched first time
   (a) $\widetilde{tag}_w \xleftarrow{\$} \{0,1\}^\lambda$
   (b) $\widetilde{T}_{tag}[w] \leftarrow \widetilde{tag}_w$

Else

(a) $\widetilde{tag}_w \leftarrow \widetilde{T}_{tag}[w]$

2. If $RO[(\widetilde{tag}_w, id_i^w, i)]$ is not null,

(a) $pos_i \leftarrow RO[(\widetilde{tag}_w, id_i^w, i)]$

Else

(a) $pos_i \leftarrow$ a random pos_i such that $\widetilde{T}'_{sig}[pos_i] = 0$

(b) $RO[(\widetilde{tag}_w, id_i^w, i)] \leftarrow pos_i$

(c) $\widetilde{T}'_{sig}[pos_i] \leftarrow 1$

3. Simulate Σ_f with $\widetilde{\tau}_{\Sigma_f} \leftarrow \mathtt{Sim}_{\Sigma_f}(\mathcal{L}_{srch}^{\Sigma_f}(w))$

4. return $\widetilde{\tau}_s^{\Psi_f} = (\widetilde{\tau}_{\Sigma_f}, \widetilde{tag}_w)$

Simulating Update token: Leakage function to add a document f with identifier id containing keyword set $\{w_1, w_2, \ldots, w_{n_w}\}$ is given by

$\mathcal{L}_{updt}^{\Psi_f}(f) = \{H'(id), \{(\mathcal{L}_{updt}^{\Sigma_f}(w_i, id)) : i = 1, 2, \ldots, n_{id}\}\}$.

1. For each keyword $w_i \in f$

(a) $\widetilde{\tau}_u^i \leftarrow \mathtt{Sim}_{\Sigma_f}(\mathcal{L}_{updt}^{\Sigma_f}(w, id))$

(b) If $\widetilde{T}_{tag}[w_i]$ is null, i.e, the keyword is searched first time

i. $\widetilde{tag}_{w_i} \xleftarrow{\$} \{0,1\}^\lambda$

ii. $\widetilde{T}_{tag}[w_i] \leftarrow \widetilde{tag}_w$

Else

i. $\widetilde{tag}_{w_i} \leftarrow \widetilde{T}_{tag}[w_i]$

(c) $c_{w_i} \leftarrow C[w_i] + 1$

(d) If $RO[(\widetilde{tag}_{w_i}, id, (c_{w_i} + 1))]$ is not null,

i. $\widetilde{pos}_i \leftarrow RO[(\widetilde{tag}_{w_i}, id, (c_v + 1))]$

Else

i. $\widetilde{pos}_i \leftarrow$ a random pos_i such that $\widetilde{T}_{sig}[pos_i]$ is null

ii. $RO[(\widetilde{tag}_{w_i}, id, (c_{w_i}+1))] \leftarrow \widetilde{pos}_i$

iii. $\widetilde{T}'_{sig}[pos_i] \leftarrow 1$

(e) $\widetilde{\sigma}_i \xleftarrow{\$} \mathsf{G}$

2. $\widetilde{pos} \leftarrow \{\widetilde{pos}_1, \widetilde{pos}_2, \ldots, \widetilde{pos}_{n_{id}}\}$

3. $\widetilde{\sigma} \leftarrow \{\widetilde{\sigma}_1, \widetilde{\sigma}_2, \ldots, \widetilde{\sigma}_{n_{id}}\}$

4. Return $\widetilde{\tau}_u^{\Psi_f} = (\widetilde{pos}, \widetilde{\sigma})$

Since, in each entry, the signature generated in T_{sig} is of the form $g^{\alpha m r}$ and corresponding entry in $\widetilde{T}_{sig}$ is of the form $g^{\alpha r'}$, where r is pseudo-random (as R is so) and r' is randomly taken, we can say that power of g in both are indistinguishable. Hence, T_{sig} and $\widetilde{T}_{sig}$ are indistinguishable.

Besides, the indistinguishability of $\widetilde{\tau}_u^{\Psi_f}$, $\widetilde{\tau}_s^{\Psi_f}$ with respect to $\tau_s^{\Psi_f}$, $\tau_u^{\Psi_f}$ respectively follows from the pseudo-randomness of F. □

5.3.2 Deletion support

Srica, i.e., Ψ_f can be extended to deletion support by duplicating it. Together with Ψ_f for addition, a duplicate Ψ'_f can be kept for deleted files. During the search, the auditor verifies both separately. The client gets results from both Ψ_f and Ψ'_f, accepts only if both are verified, and gets the final result calculating the difference.

5.4 Performance evaluation

Our generic VSSE Aris requires only one hash-value computation to verify a search which is optimal. Again, during building, the owner requires $2|\mathcal{W}|$ extra hash-value computation twice of the optimal. We can take that much computation to protect the scheme from malicious cloud server without any extra client storage.

We have compared our verifiable DSSE scheme Srica with verifiable dynamic schemes by Yoneyama and Kimura [107], Bost and Fouque [17], Miao et al. [66], Zhu et al. [112] and Jiang et al. [44]. The comparison is shown in Table 5.2. From the table, it can be observed that Srica is very efficient with respect to low resource owner. Extra computation needed by the owner, to verify the search, is only $|R_w|$ multiplication which is very less than the others. The owner also does not require any more storage than the built-in forward private DSSE scheme.

Table 5.2: Comparison of verifiable dynamic SSE schemes

Scheme Name	Forward privacy	Public verifiability	Extra Storage		Extra Computation			Extra Communication	
			owner	cloud	owner	cloud	auditor	owner	auditor
Yoneyama and Kimura [107]	✓	×	$O(\|\mathcal{W}\|)$	$O(\|\mathcal{W}\|log\|\mathcal{DB}\|)$	$O(\|R_w\|)$	$O(\|R_w\|)$	–	$O(1)$	–
Bost and Fouque [17]	×	×	$O(\|\mathcal{W}\|)$	$O(\|\mathcal{W}\|)$	$O(\|R_w\|)$	$O(1)$	–	$O(1)$	–
Miao et al. [66]	×	✓	$O(\|\mathcal{W}\|)$	$O(N+\|\mathcal{W}\|)$	$O(\|R_w\|)$	$O(\|R_w\|)$	–	$O(1)$	–
Zhu et al. [112]	×	×	$O(1)$	$O(1)$	$O(\|R_w\|)$	$O(\|R_w\|+N)$	–	$O(\|R_w\|)$	–
Jiang et al. [44]	×	✓	$O(1)$	$O(\|\mathcal{W}\|)$	$O(\log\|\mathcal{W}\|)$	$O(\|R_w\|+N)$	–	$O(1)$	–
Srica	✓	✓	$O(1)$	$O(N)$	$O(\|R_w\|)$	$O(\|R_w\|)$	$O(1)$	$O(1)$	$O(1)$

Where N is the #keyword-doc pairs. Here extra storage is calculated over all storage, extra communication and computation are for a single search.

Conclusion

In this chapter, we have proposed a privately verifiable SSE scheme and a publicly verifiable DSSE scheme. However, the scheme is only for single keyword search queries. There are many other complex queries too. In the next chapter, we study conjunctive keyword search. We try to design a scheme that is forward private and verifiable.

Chapter 6

Forward Private and Verifiable Conjunctive Search Scheme

Sensitive data is often encrypted before storing it in outsourced servers (clouds). This makes searching difficult. In this chapter, we consider the problem of keyword search. As we have seen in Chapter 4 and Chapter 5, a searchable encryption (SE) scheme allows a cloud server to search on encrypted data and return the results of the search query to the client.

In the previous chapter, we have addressed verifiable single keyword DSSE. In this chapter, we will study a conjunctive SE scheme with both forward privacy and verifiability. We present a generic scheme that converts any forward private DSE scheme to a verifiable conjunctive DSE scheme preserving its forward privacy. For verifiability, we propose a new cryptographic accumulator called *dynamic interval accumulation tree (DIA Tree)*.

Cryptographic accumulators are used for proving membership as well as non-membership of elements in a set. When the size of the set is large, proof generation and (or) proof size becomes expensive. Though the existing accumulator scheme like [5] can build an accumulation tree for a static database that can provide the proof of membership as well as non-membership, it is inefficient for a dynamic set. Papamanthou et al. [76] presented a scheme that dealt with the dynamic set that generates membership proofs efficiently. They extended their scheme with an additional authenticated tree that allows non-membership checks. However, this additional structure does not support updates.

In this chapter, we have proposed a **Dynamic Interval Accumulation tree** (DIA tree) that efficiently works for both membership and non-membership witnesses and returns proofs even on large dynamic dataset efficiently. We have used the DIA Tree in our conjunctive DSE scheme for verifiability. Please note that DIA trees are of independent interest and can be applied to the other applications not just constructing DSE schemes.

Our contributions In this work, we make the following contributions.

- We propose an accumulator using a new data structure called Dynamic Interval Accumulation tree (DIA tree) that supports efficient proofs of membership and non-membership. To the best of our knowledge, there is no previous scheme in the literature that provides a single

authentication data structure supporting both membership and non-membership proofs efficiently together with update support. We provide formal security proof for the accumulator.

- We propose a generic verifiable conjunctive search DSE scheme Blasu that converts any forward private conjunctive DSE scheme to a verifiable forward private conjunctive DSE scheme, without losing its forward privacy property and without using any extra client storage for verifiability. To the best of our knowledge, our proposed scheme is *the first forward private as well as verifiable conjunctive SE scheme in a dynamic setting*. Moreover, we have given security proof of the scheme. We have shown that the proposed scheme Blasu is secure against adaptive chosen query attack.

- We compare our proposed scheme Blasu with the existing schemes and show that the scheme is practical.

Organization We summarize our work in the chapter as follows. We briefly introduce the required cryptographic tools in Section 6.1. We propose an authenticated data structure DIA tree in Section 6.2, together with its security proof. Using the tree, in Section 6.3, we propose a DSE scheme Blasu that provides verifiability without extra client storage. In Section 6.4, we compare our scheme Blasu with a few of the existing similar schemes.

6.1 Preliminaries

6.1.1 System model

In our model of conjunctive verifiable DSE, there are three entities– client, cloud and auditor. The system model is shown in Figure 6-1. Here, we briefly describe the system model as follows.

- **Client:** The *client* owns the database and requires outsourcing its data. It is assumed to be a *trusted* party. Before outsourcing the data, it builds a secure search index. Then, it encrypts and sends the data together with the index. It is the user of the database as well. For every query made, it generates an encrypted query token and sends it to the cloud. Finally it receives the result from the cloud.

- **Cloud:** The entity *cloud* is assumed to be malicious. It provides both storage and computation services. It stores the encrypted data. When a search query is given, it computes over

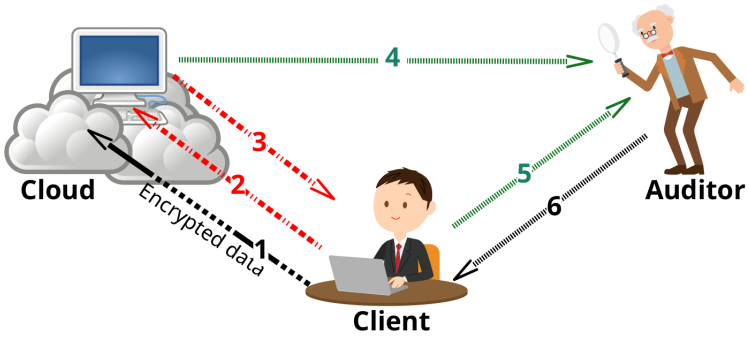


Figure 6-1: The system model of a conjunctive verifiable DSE scheme

1. Client sends encrypted data, 2. Client sends query token to the Cloud, 3. After searching, Cloud sends Result, 4. Cloud sends proof to the Auditor, 5. Client sends proof of received result, 6. Auditor sends verification result.

the data and returns result to the client. It also sends proof of its correct execution to the auditor.

- **Auditor:** The entity *auditor* is an authority that tells, verifying the proof received from the client and the cloud, whether the returned result is correct or not. Any party, including the client, can be an auditor.

6.1.2 Design goals

While designing such a scheme, assuming the above system model and aiming to provide a solution toward a verifiable conjunctive search DSE scheme, with keeping it forward private, we achieve confidentiality, scalability, efficiency and update support over the encrypted outsourced data, as described in Section 1.1.2.

Moreover, Since a DSE scheme, without forward privacy, is vulnerable to even honest-but-curious adversary, it is desirable the scheme to be forward private while achieving public verifiability for a conjunctive search result.

6.1.3 Definitions and terminologies

6.1.3.1 Notations

Let identifiers of the documents belong to the space of document identifiers $\mathcal{D}$. Let $\mathcal{DB} \subseteq \mathcal{D}$. Let each document contains some keywords belonging to a keyword space $\mathcal{W}$. For each keyword $w \in \mathcal{W}$, let $DB(w) = \{id_1^w, id_2^w, \ldots, id_{n_w}^w\}$ be the set of document identifiers that contains w, where $n_w = |DB(w)|$ and $id_i^w \in \mathcal{DB}$ is the ith identifier. n_w is also called the *frequency of the keyword* $w \in W$. Thus, $\bigcup_{w \in \mathcal{W}} DB(w) \subseteq \mathcal{DB}$.

Let $EDB = \{c_{id} : id \in \mathcal{D}\}$ where by c_{id} we mean the encrypted document that has id as identifier. Let us consider the existence of a one-way function which maps every document identifier to a random number. However, when we say cloud returns documents to the client, we assume the cloud performs the function on every identifier before returning them.

Let, $R : \{0,1\}^* \to \{0,1\}^*$ be a PRNG and $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ be a PRP.

In Table 6.1, we have shown some notations used in this chapter.

Table 6.1: Notations used in our conjunctive verifiable search scheme

Symbol	Meaning	Symbol	Meaning
S	Set of elements from $\{0,1\}^*$	DT	DIA tree
B_i	ith bucket	d	root of DT
S_i	sorted set of elements in B_i	s	secret key of client for DT
l_i	lower bound of B_i	a_i	ith accumulators
r_i	upper bound of B_i	$\mathcal{W}$	the keyword set
S_i^o	$S_i \cup \{l_i, r_i\}$	w_i	ith keyword in $\mathcal{W}$
G	a bilinear group	x, x', x''	elements of G
$\hat{e}$	a bilinear map	$wt(x)$	membership witness of x
e	an elements from $\{0,1\}^*$	$wtn(x)$	non-membership witness of x
g	a generator of G	id_j^w	jth file that contains w
$\mathcal{H}$	a one-way hash	$\phi(v)$	membership proof of v
	$\{0,1\}^* \to \mathsf{G} \setminus \{1\}$	$SL(v)$	set of siblings of v

6.1.3.2 Dynamic Searchable Encryption (DSE) scheme

A dynamic searchable encryption (DSE) scheme Σ consists of algorithms (`KeyGen`, `Build`, `SrchTkn`, `Search`, `UptdTkn`, `Update`), between a client and a cloud server, briefly described as follows.

$K_\Sigma \leftarrow \texttt{KeyGen}(1^\lambda)$: is a PPT algorithm run by the client that takes a security parameter 1^λ and outputs the secret key K_Σ.

$(\xi, EDB) \leftarrow \texttt{Build}(\mathcal{DB}, K_\Sigma)$: is a client-side PPT algorithm that takes the dataset and the secret key as input and outputs a pair (ξ, EDB) where EDB the encrypted database, and ξ an encrypted index.

$\tau_w^\Sigma \leftarrow \texttt{SrchTkn}(w, K_\Sigma)$: is also a client-side PPT algorithm that generates an encrypted search trapdoor τ_w^Σ for a keyword w with the help of K_Σ.

$R_w \leftarrow \texttt{Search}(\xi, \tau_w^\Sigma)$: with this PPT algorithm, the cloud server perform search over ξ for τ_w^Σ and returns the search result R_w to the client.

$\tau_u^\Sigma \leftarrow \texttt{UptdTkn}(K_\Sigma, w, id)$: Given a keyword-document pair (w, id) the client generates an token, encrypted with K_Σ, for update with the help of this PPT algorithm.

$\xi' \leftarrow \texttt{Update}(\xi, \tau_u^\Sigma, op)$ is a cloud-side algorithm that updates ξ according to the op for the update token τ_u^Σ, and keeps the updated index ξ'.

Definition 6.1 (CKA2-security of a DSE scheme). *[48] Let* $\Sigma =$(`KeyGen`, `Build`, `SrchTkn`, `Search`, `UptdTkn`, `Update`) *be a DSE scheme. Let* $\mathcal{A}$ *be a stateful adversary,* $\mathcal{C}$ *be a challenger,* $\mathcal{S}$ *be a stateful simulator and* $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{bld}, \mathcal{L}_\Sigma^{srch}, \mathcal{L}_\Sigma^{updt})$ *be a stateful leakage algorithm. Let us consider the following two games.*

Real$_\mathcal{A}^\Sigma(\lambda)$*: At first* $\mathcal{C}$ *generates a key* $K_\Sigma \leftarrow \texttt{KeyGen}(1^\lambda)$*. In the same time,* $\mathcal{A}$ *chooses a set of documents* $\mathcal{DB}$ *and sends it to* $\mathcal{C}$*. Then,* $\mathcal{C}$ *computes* $(\xi, EDB) \leftarrow \texttt{Build}(\mathcal{DB}, K_\Sigma)$ *and sends* (ξ, EDB) *to* $\mathcal{A}$*. During search phase,* $\mathcal{A}$ *makes a polynomial number of adaptive queries. In each query* $\mathcal{A}$ *sends either a search query for a keyword* w *or an update query for* (id, op) *for a document with identifier* id *operation* op*. Depending on the query,* $\mathcal{C}$ *returns either the search token* $t_w^\Sigma \leftarrow \texttt{SrchTkn}(w, K_\Sigma)$ *or the update token* $t_u^\Sigma \leftarrow \texttt{UptdTkn}(K_\Sigma, w, id)$ *to* $\mathcal{A}$*. Finally* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

Ideal$_{\mathcal{A},\mathcal{S}}^\Sigma(\lambda)$*: At first,* $\mathcal{A}$ *generates* $\mathcal{DB}$ *and gives it to* $\mathcal{S}$ *together with and* $\mathcal{L}_\Sigma^{bld}(\mathcal{DB})$*. On receiving* $\mathcal{L}_\Sigma^{bld}(\mathcal{DB})$*,* $\mathcal{S}$ *generates* (ξ, EDB) *and sends it to* $\mathcal{A}$*.* $\mathcal{A}$ *makes a polynomial number of adaptive queries* $q \in \{w, (id, op)\}$*. For each query,* $\mathcal{S}$ *is given either* $\mathcal{L}_\Sigma^{srch}(w)$ *or* $\mathcal{L}_\Sigma^{updt}(id, op)$*. Depending on the query* q*,* $\mathcal{S}$ *returns to* $\mathcal{A}$ *either search token* t_w^Σ *or update token* t_u^Σ*. Finally* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

We say Σ *is* $\mathcal{L}_\Sigma$*-secure against adaptive dynamic chosen-keyword attacks if for any PPT (prob-*

abilistic polynomial-time) adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that

$$|Pr[\textbf{\textit{Real}}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - Pr[\textbf{\textit{Ideal}}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mu(\lambda) \tag{6.1}$$

Correctness: The correctness of a DSE scheme ensures that the every search protocol must return the correct result for every query, except with negligible probability.

A DSE scheme Σ that does not leak any information about the previous search results, is said to be **forward private**.

The schemes [17], [16], etc., are good examples of a forward private DSE schemes. In our proposed scheme, we use any forward private DSE scheme Σ as a black box. We assume the black box scheme Σ is correct and $\mathcal{L}_{\Sigma}$-secure against adaptive dynamic chosen-keyword attacks.

6.1.3.3 Verifiable Dynamic Conjunctive Searchable Encryption (VDCSE)

A *dynamic conjunctive SE (DCSE)* scheme supports conjunctive keyword search in dynamic database, i.e., given a set of keywords, it returns the set of documents containing all keywords. In the presence of a malicious adversary, a *verifiable dynamic conjunctive SE* scheme provides the ability to verify whether the returned result is consistent with the updated database. After a search is performed, both the client and client give their part of the proof. An auditor or any third party can verify the result from the proofs. We define a VDCSE scheme formally as follows. A *dynamic conjunctive SE (DCSE)* scheme supports conjunctive keyword search in dynamic database. In the presence of a malicious adversary, a *verifiable dynamic conjunctive SE* scheme provides the ability to verify whether the returned result is consistent with the updated database. We define a VDCSE scheme formally as follows.

Definition 6.2 (Verifiable Dynamic Conjunctive Searchable Encryption)**.** *A verifiable dynamic conjunctive searchable encryption (VDCSE) scheme Ψ is a tuple* (`VCKeyGen`, `VCBuild`, `VCSrchTkn`, `VCSearchCD`,`VCSearchCT`, `VCUpdtTkn`, `VCUpdate`) *of algorithms defined as follows.*

- $K_{\Psi} \leftarrow$ `VCKeyGen`(λ)*: Given a security parameter λ, this PPT algorithm, run by the client, outputs a key K_{Ψ}.*
- $(\mathtt{st}, EDB, \gamma, \mathcal{I}\ ) \leftarrow$ `VCBuild`$(\mathcal{DB}, K_{\Psi})$*: This is PPT algorithm run by the client. Given K_{Ψ} and a set of documents $\mathcal{DB}$, it outputs the encrypted set of documents EDB together with an encrypted search index γ and an auxiliary data structure $\mathcal{I}$ for verifiability. It also outputs state* `st` *of the database.*

- $\tau_s^{\Psi} \leftarrow \texttt{VCSrchTkn}(K_{\Psi}, \hat{w}, \texttt{st})$*: Given* K_{Ψ}*,* $\texttt{st}$ *and a set of keywords* $\hat{w}$*, the client runs this PPT algorithm and outputs a search token* τ_s^{Ψ}*.*

- $(pf_c, \hat{R}_{\hat{w}}, X_{\hat{w}}) \leftarrow \texttt{VCSearchCD}(\gamma, \mathcal{I}, t_s^{\Psi})$*: Given* γ*,* $\mathcal{I}$ *and* τ_s^{Ψ}*, in this cloud-run PPT algorithm, the cloud returns result set* $\hat{R}_{\hat{w}}$ *of document ids, a proof* pf_c*.*

- $(pf_u) \leftarrow \texttt{VCSearchCT}(\hat{R}_{\hat{w}}, t_s^{\Psi})$*: Given* $\hat{R}_{\hat{w}}$*, in this client-run PPT algorithm, the client returns a proof* pf_u*.*

- $\nu_{\hat{w}} \leftarrow \texttt{VCVerify}(d, pf_u, pf_c, \hat{R}_{\hat{w}})$ *This is a PPT algorithm that takes the proofs* pf_u*,* pf_c *together with the result* $\hat{R}_{\hat{w}}$ *and outputs the verification bit* $\nu_{\hat{w}}$

- $(\tau_u^{\Psi}, \texttt{st}') \leftarrow \texttt{VCUpdtTkn}(K, id, \texttt{st})$*: Taking a key* K_{Ψ}*, a document identifier* id *and the present state* $\texttt{st}$*, the cloud runs this PPT algorithm and outputs an update token* τ_u^{Ψ} *and a new state* $\texttt{st}'$*.*

- $(EDB', \gamma', \mathcal{I}') \leftarrow \texttt{VCUpdate}(\gamma, EDB, \mathcal{I}, \tau_u^{\Psi}, op)$*: It is a cloud-run PPT algorithm which takes an update token* τ_u^{Ψ}*, operation bit* op*,* EDB*, the index* γ *and the auxiliary information* $\mathcal{I}$ *and outputs updated* $(EDB', \gamma', \mathcal{I}')$*.*

Correctness A VDCSE scheme Ψ is said to be *correct* if $\forall \lambda \in \mathsf{N}, \forall K_{\Psi}$ generated using $\texttt{KeyGen}(\lambda)$ and all sequences of search and update operations, every search outputs the correct set of identifiers, except with a probability $neg(\lambda)$.

Verifiability By verification, we mean verification of a search result. We verify whether the search is performed on the current state of the database. We do not include verification of updates on the cloud-side. If the cloud cheats, and updates incorrectly, it will fail verification test when a search result includes such updated information.

6.1.3.4 Security definitions

We follow security definition of [85]. Security of a VDCSE scheme is divided into two parts– confidentiality and soundness, described as follows. Confidentiality ensures the cloud does not get any extra meaningful information other than this is allowed. On the other hand, the soundness property ensures that the client gets the correct result. If some malicious cloud wants to return an incorrect result, the verifier can detect it.

Confidentiality: We have considered the cloud to be malicious. It can misuse the data if it can get information about the actual data. So, in a VDCSE, the cloud server should not know more

information than it is allowed for. This property protects the client to leak only allowed amount of information, not more than that. We define the confidentiality of a VDCSE as follow.

Definition 6.3 (CKA2-confidentiality of a VDCSE scheme). *Let* $\Psi = (\texttt{VCKeyGen}, \texttt{VCBuild}, \texttt{VCSrchTkn}, \texttt{VCSearchCD}, \texttt{VCSearchCT}, \texttt{VCUpdtTkn}, \texttt{VCUpdate})$ *be a verifiable dynamic conjunctive searchable Encryption scheme. Let* $\mathcal{A}$*,* $\mathcal{C}$ *and* $\mathcal{S}$ *be a stateful adversary, a challenger and a stateful simulator respectively. Let* $\mathcal{L}_\Psi$=$(\mathcal{L}_\Psi^{bld}, \mathcal{L}_\Psi^{srch}, \mathcal{L}_\Psi^{updt})$ *be a stateful leakage algorithm. Let us consider the following two games.*

Real$_{\mathcal{A}}^{\Psi}(\lambda)$*: At first, a key* $K_\Psi \leftarrow \texttt{VCKeyGen}(\lambda)$ *is generated by the challenger* $\mathcal{C}$*. Then a database* $\mathcal{DB}$ *is chosen by the adversary* $\mathcal{A}$ *and sent to* $\mathcal{C}$*. The encrypted database* EDB *is built and an encrypted search index is generated by* $\mathcal{C}$ *as* $(\texttt{st}, EDB, \gamma, \mathcal{I}) \leftarrow \texttt{VCBuild}(\mathcal{DB}, K_\Psi)$ *and then* $(EDB, \gamma, \mathcal{I})$ *is sent to* $\mathcal{A}$*. In the next phase a polynomial number of adaptive queries are made by* $\mathcal{A}$*. In each of them, either a search query for a keyword set* $\hat{w}$ *or an update query for a keyword-document pair* (w, id) *and operation bit* op *is sent to* $\mathcal{C}$ *by it. In sequence,* $\mathcal{C}$ *returns either a search token* $\tau_s^\Psi \leftarrow \texttt{VCSrchTkn}(K_\Psi, \hat{w}, \texttt{st})$ *or an update token* $\tau_u^\Psi \leftarrow \texttt{VCUpdtTkn}(K, id, \texttt{st})$ *to* $\mathcal{A}$*. Finally, a bit* b*, that is output of the experiment, is returned by* $\mathcal{A}$*.*

Ideal$_{\mathcal{A},\mathcal{S}}^{\Psi}(\lambda)$*: At first, a database* $\mathcal{DB}$ *is chosen by* $\mathcal{A}$ *and is given to* $\mathcal{S}$ *together with* $\mathcal{L}_\Psi^{bld}(\mathcal{DB})$*. Then, a simulated database and index* (EDB, γ) *is generated by* $\mathcal{S}$ *and sent to* $\mathcal{A}$*. Then a polynomial number of adaptive queries are made by* $\mathcal{A}$*. For each query, either* $\mathcal{L}_\Psi^{srch}(\hat{w}, \mathcal{DB})$ *or* $\mathcal{L}_\Psi^{updt}(op, w, id)$*, depending on the query, is given to* $\mathcal{S}$*. Accordingly,* $\mathcal{S}$ *returns either search token* τ_s^Ψ *or update token* τ_u^Ψ *to* $\mathcal{A}$*. Finally, a bit* b'*, that is output of the experiment, is returned by* $\mathcal{A}$*.*

We say Ψ *is* $\mathcal{L}_\Psi$*-secure against adaptive dynamic chosen-query attacks if* $\forall$ *PPT adversary* $\mathcal{A}$*,* $\exists$ *a simulator* $\mathcal{S}$ *such that*

$$|Pr[\textbf{\textit{Real}}_{\mathcal{A}}^{\Psi}(\lambda) = 1] - Pr[\textbf{\textit{Ideal}}_{\mathcal{A},\mathcal{S}}^{\Psi}(\lambda) = 1]| \leq \mu(\lambda) \tag{6.2}$$

where $\mu(\lambda)$ *is negligible in* λ*.*

Soundness Since, the cloud has to provide computational as well as storage resources which has a good amount of monetary cost, the cloud server can skip by giving incomplete or incorrect result. In such a scenario, the soundness property ensures the client gets complete result with respect to the present state of database. The game-based definition of soundness property of a VDCSE scheme is given as follow.

Definition 6.4 (Soundness of a VDCSE scheme). *Let Ψ be a VDCSE scheme with $\Psi = (\texttt{VCKeyGen}, \texttt{VCBuild}, \texttt{VCSrchTkn}, \texttt{VCSearchCD}, \texttt{VCSearchCT}, \texttt{VCUpdtTkn}, \texttt{VCUpdate})$. Let us consider the following game.*

sound$^{\Psi}_{\mathcal{A}}(\lambda)$*: At first, a key $K_\Psi \leftarrow \texttt{VCKeyGen}(\lambda)$ is generated by the challenger $\mathcal{C}$. Then, a database $\mathcal{DB}$ is chosen by the adversary and sent to $\mathcal{C}$. The encrypted database is computed as $(\texttt{st}, EDB, \gamma, \mathcal{I}) \leftarrow \texttt{VCBuild}(\mathcal{DB}, K_\Psi)$ by $\mathcal{C}$ and then $(EDB, \gamma, \mathcal{I})$ is sent to $\mathcal{A}$. A polynomial number of adaptive queries are made by $\mathcal{A}$. In each of them, either a search query, for a keyword set $\hat{w}$, or an update query, for a keyword-document pair (w, id) and operation bit op, is sent to $\mathcal{C}$. In response, depending on the query, either a search token $\tau_s^\Psi \leftarrow \texttt{VCSrchTkn}(K_\Psi, \hat{w}, \texttt{st})$ or an update token $\tau_u^\Psi \leftarrow \texttt{VCUpdtTkn}(K, id, \texttt{st})$ is returned to $\mathcal{A}$.*

In the challenge phase, a target keyword set $\hat{w}$ is chosen by $\mathcal{A}$ and a search query for $\hat{w}$ is sent to $\mathcal{C}$. In response, a search token τ_s^Ψ is returned from which $(R_{\hat{w}}, \nu_{\hat{w}})$ is searched by $\mathcal{A}$, where $\nu_{\hat{w}} = accept$ is verification bit from $\mathcal{C}$. Finally, a pair $(R^*_{\hat{w}}, \nu^*_w)$ for a keyword set $\hat{w}$ is generated by $\mathcal{A}$. If $\nu^*_{\hat{w}} = accept$ even when $R^*_{\hat{w}} \neq \bigcap_{w \in \hat{w}} DB(w)$, $\mathcal{A}$ returns 1 as output of the game, otherwise returns 0.

We say that Ψ is sound *if $\forall$ PPT adversaries $\mathcal{A}$, $Pr[$**sound**$^{\Psi}_{\mathcal{A}}(\lambda) = 1] \leq \mu(\lambda)$.*

6.2 Dynamic Interval Accumulation tree (DIA Tree)

If we consider membership witnesses, they cannot be updated without the secret key. However, the client computes them initially by itself and stores them in the cloud. It fetches and updates them each time a new element is added or deleted. For a given set of elements, if only one accumulator is generated for the set, then the generating membership witness for a new element x becomes inefficient. This is because the membership witness generation considers all elements belong to the set in the computation i.e., computational complexity grows with the size of the set. If the set is too large, the computation of the witness becomes impractical.

This is why, instead of generating a single accumulator, the set is divided into buckets and a separate accumulator is generated for every bucket. In the next level, this set of accumulators becomes input set and another set of accumulators is generated for them. The process continues until only one element, i.e., the root is left. The generated tree is an accumulation tree.

Papamanthou et al. [76] studied the above approach previously in their work of authenticated hash table. Though this scheme is good for membership-proof and supports updates, it has a serious issue. For non-membership poof, the scheme considers an additional accumulation tree which is

based on intervals. This tree does not support deletion. This makes the tree an append-only tree.

We take a different interval approach and construct DIA tree. The tree gives the ability to perform both membership and non-membership in a single tree, even in case the set is dynamic and large.

Before describing the construction of diatree formally, we give an example.

6.2.1 Example of a DIA tree

Let us consider a 3-ary tree with height $h = 3$. Then there are $h + 1$ levels where Level-0 is the root. Then Level-2 has 9 elements. Each node at Level-2 can hold at most 5 elements. Then, all possible elements can be mapped in [0,44]. The ith element at Level-2 corresponds to the bucket $[5i, 5(i+1)-1]$. However, for construction, we want them to be in some open interval which allows any operation to effect one bucket only. So, we take ith interval as $I_i = (l_i, r_i) = (5i - 1, 5(i + 1))$ (see Figure 6-2).

Now, given a set $S = \{6, 7, 9, 13, 21, 24\}$, we consider the following 15 (open) intervals, $(-1, 5)$, $(4, 6)$, $(6, 7)$, $(7, 9)$, $(9, 10)$, $(9, 13)$, $(13, 15)$, $(14, 20)$, $(19, 21)$, $(21, 24)$, $(24, 25)$, $(24, 30)$, $(29, 35)$, $(34, 40)$, $(39, 45)$. Let I_i be the ith interval. Then we take hash $x_i = \mathcal{H}(I_i)$, for each i, to map them as an element of G. Then the 2nd Level-2 node stores $\{x_2, x_3, x_4, x_5\}$, 5th Level-2 node stores $\{x_9, x_{10}, x_{11}\}$, 6th stores x_{12} etc.

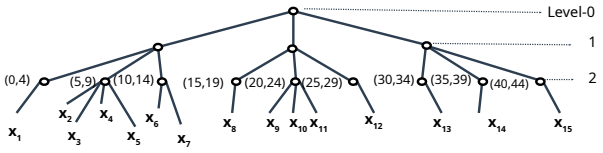


Figure 6-2: DIA tree for the set S

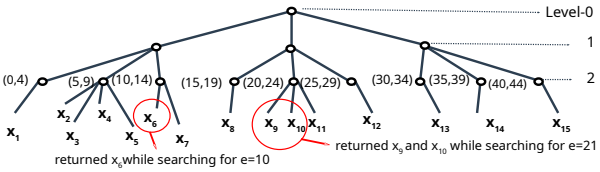


Figure 6-3: search for $e = 21$ and $e = 10$

In addition, any node, except leaves, stores accumulator for its set of children. Moreover, any node, except the root, stores witness of membership in the parent node.

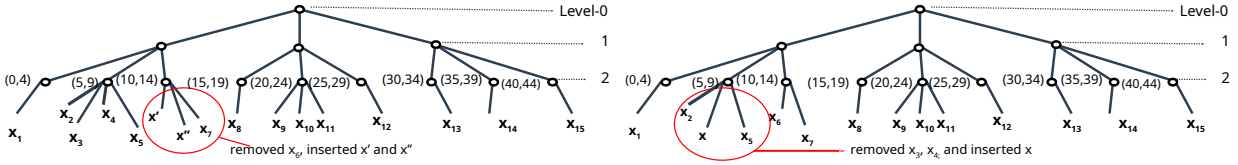
Search: To search an element $e = 21$, at first, its corresponding interval is searched. Let its boundaries be $l = 19$ and $r = 25$. And then it searches two elements $e' = 13$ and $e'' = 24$ in S such that $e' < e < e''$. Finally, it sets $e' = max\{e', l\}$ and $e'' = min\{e'', r\}$. This is equivalent to say that we are choosing two elements e' and e'' in the left and the right of e respectively from the bucket corresponding to e (see Figure 6-3).

Since, 21 is in S, it considers the intervals as $I' = (e', e) = (19, 21)$ and $I'' = (e, e'') =$

$(21, 24)$. Then calculate the hashes $x' = \mathcal{H}(I')$ and $x'' = \mathcal{H}(I'')$. We see that both x' and x'' are in the tree. So, for each of them, with the result, cloud returns proof of their existence. The proof contains accumulators and witnesses stored in every node from leaf to root in the path of the bucket.

Similarly, if 10 is searched, the proof for the interval $(9, 13)$ is returned. This is because if some element belongs to S, it appears in two intervals– in one as the right boundary and in another as the left boundary. When it is not in S, there exists an interval that contains the searched element.

Update: When we want to add $e = 11$, we find $e' = 9$ and $e'' = 13$, such that, $e' < e < e''$, in the bucket corresponding to 3 (see Fig. 6-4a). Then we just remove x_6 corresponding to the interval $I = (9, 13)$ and then add two intervals $I' = (9, 11)$ and $I'' = (11, 13)$. For that we remove $x_6 = \mathcal{H}(I)$ and add both $x' = \mathcal{H}(I')$ and $x'' = \mathcal{H}(I'')$. After doing the same, accumulators in the path of the bucket from leaf to root and and witnesses for each of their children are updated.



(a) Adding $e = 11$ (b) Deleting $e = 7$

Figure 6-4: Updating the tree

Again, we delete an element only if it exists in S. So, in that case, given an element $e = 7$, we can find two intervals I' and I'' where e is left and right bound i.e., $I' = (6, 7)$ $I'' = (7, 9)$. So, $e' = 6$ and $e'' = 9$ (see Fig. 6-4b). Let $I = (e', e'') = (6, 9)$. To delete the element $e = 7$, we remove both $x' = \mathcal{H}(I')$ and $x'' = \mathcal{H}(I'')$, and then add $x = \mathcal{H}(I)$ and update the tree accordingly.

6.2.2 DIA Tree construction

For a given set S, in our proposed DIA tree construction, the set is stored separately. A tree is constructed to give proof of whether an element exists in the given set S. We describe a DIA tree scheme $\mathtt{DIAT}$ as a tuple $(\mathtt{Init}, \mathtt{BuildTree}, \mathtt{Search}, \mathtt{Update})$ of algorithms as follows.

Initialization $[(s, tup) \leftarrow \mathtt{DIAT.Init}(1^\lambda)]$: Given a security parameter λ, a tuple $(p, \mathsf{G}, \mathsf{G}, \hat{\mathsf{e}}, \mathsf{g}) \leftarrow \mathtt{BMGen}(1^\lambda)$ is generated. Let $tup = (p, \mathsf{G}, \mathsf{G}, \hat{e}, g)$. An element s is chosen randomly from Z_p^* and finally (tup, s) is returned.

Building the Tree $[(DT, d) \leftarrow \texttt{DIAT.BuildTree}(tup, s, S)]$: We consider in the given set $S = \{e_1, e_2, \ldots, e_n\}$, each e_i is of fixed length and sorted. Let the complete range of elements be $[0, 2^\lambda)$. We divide the range into b half-open intervals, each of size 2^η. Then the number of intervals will be $b = 2^{\lambda-\eta}$. Let the ith intervals be $[2^{i-1}, 2^i)$ and it corresponds to the ith bucket B_i. Finally, we take range of bucket B_i as closed intervals $[2^{i-1} - 1, 2^i]$. Let $S_i = \{e_{i,1}, e_{i,2}, \ldots, e_{i,n_i}\}$ be the sorted set of elements from S, falls into the bucket B_i. We consider $S_i^o = S_i \cup \{l_i, r_i\}, i = 1, \ldots, b$, where $l_i = 2^{i-1} - 1$, $r_i = 2^i$, $\forall i = 1, \ldots, b$, and $l_0 = -\infty$ and $r_{n_i} = +\infty$. Now, we treat each S_i^o separately as follow.

Let $I_{i,j} = (e_{i,j}, e_{i,j+1}), \forall j = 0, \ldots, n_i$, where $e_{i,0} = l_i$ and $e_{i,n_i} = r_i$. Then, we map each of the intervals in G as $x_{i,j} = \mathcal{H}(I_{i,j})$, where $\mathcal{H} : \{0,1\}^* \rightarrow \mathsf{G}$ that brings each interval to an element in G. Let $\bar{S}_i = \{(x_{i,j}) : j = 0, \ldots, n_i\}$. Thus, the set of elements belongs to S_i transfers to the set of $\bar{S}_i$.

Finally, we get the sets $\bar{S}_1, \bar{S}_2, \ldots, \bar{S}_b$ and make accumulators for each of them. The elements of the set $\bar{S}_i$ are kept in the leaves, say at Level-h, where h is the height of the tree. For each set $\bar{S}_i$, the accumulator is generated as $a_i \leftarrow \texttt{AC.Gen}(\bar{S}_i, s)$ where $a_i = g^{\prod_{x \in \bar{S}_i}(\mathcal{H}'(x)+s)} \in \mathsf{G}$ and $\mathcal{H}' : \mathsf{G} \rightarrow \mathsf{Z}_\mathsf{p}^*$.

Next, we start from the set $\{a_1, a_2, \ldots, a_b\}$, recursively make an m-ary tree, above the leaves, until we reach only one accumulator, say d, the digest of the tree. If h is the height of the tree then $m^{h-1} = b$. Thus, every internal node of the tree DT stores the accumulator corresponding to its set of children. Moreover, every node, including leaf nodes and excluding the root, contains membership proof with respect to its parent. Thus if v is a node, it keeps membership proof as $\phi(v) = g^{\prod_{x \in SL(v)}(\mathcal{H}'(x)+s)} \in \mathsf{G}$ where $SL(v)$ is the set of siblings of v. We can see that the each internal node has same number of children, whereas Level-$(h-1)$ nodes stores random number of children. Finally, the accumulation tree DT is returned to the cloud, and the client stores d.

Search $[(b_x, w_e) \leftarrow \texttt{DIAT.Search}(DT, e)]$: Given an element e, this algorithm tells whether it exists together with a witness. This is done as follows.

Let B_k be the bucket for e. If $e \notin S_k$, it finds other two elements $e', e'' \in S_k^o$ such that $e' < e < e''$ in the bucket corresponding to e. Then it computes $x \leftarrow \mathcal{H}(I)$ where $I = (e', e'')$. Then it gives membership witness $wt(x) = \pi_I$ for x in DT. Let $v_0, v_1, \ldots, v_h$ nodes in the path corresponding to the node x where v_h is the root of the tree. Then, $wt(x)$ is of the form (e', e'', π_I) where $\pi_I = (\pi_1, \pi_2, \ldots, \pi_h)$ and each π_i is a pair (α_i, β_i) defined as

$$\alpha_i = \Phi(v_{i-1}) \text{ and } \beta_i = g^{\prod_{u \in SL(v_{i-1})}(\mathcal{H}(\Phi(u))+s)}, i = 1, 2, \ldots, h \tag{6.3}$$

Note that, in our proposed scheme, we pre-compute both α_i and β_i.

Now, if $e \in S_k$, it finds another two elements $e', e'' \in S_k^o$ such that $e' < e < e''$. Then the witness of non-existence of x is given by $wtn(x) = (e', e'', \pi_{I'}, \pi_{I''})$ where $I' = (e', e)$ and $I'' = (e, e'')$.

$b_e = 1$ indicates existence and $w_e = wt(x)$ is set whereas $b_x = 0$ indicates the opposite and $w_e = wtn(x)$ is set.

Verify Search [$b \leftarrow$ `DIAT.VerifySearch`(d, b_e, w_e, e)]: If $b_e = 1$, verifier verifies whether $\beta_i^{\mathcal{H}(\alpha_{i-1})+s} = \alpha_i, \forall i = 1, 2, \ldots, h$. It recomputes the element $x = \mathcal{H}'(e', e'')$ and computes $\alpha_i = x$. Then, it verifies

$$\hat{e}(\alpha_i, g) = \hat{e}(\beta_{i-1}, g^{(\mathcal{H}(\alpha_{i-1})+s)}), i = 1, 2, \ldots, h,$$

Additionally, it is checked if $\hat{e}(d, g) = \hat{e}(\beta_l, g^{(\mathcal{H}(\alpha_h)+s)})$ where d is the root digest. The result is accepted if all are verified correctly.

If $b_e = 0$, it recomputes the element $x_1 = \mathcal{H}'(e', e)$ and $x_2 = \mathcal{H}'(e, e'')$. verifies the same for both intervals. It returns accept if, witnesses are verified for both intervals. verification is done similarly as above.

Update [$d' \leftarrow$ `DIAT.Update`(DT, T, s, d, e, op)]: Given an element e let B_k be its bucket. Then it finds two elements $e', e'' \in S_k^o$, such that $e' < e < e''$. Then $x' \leftarrow \mathcal{H}'(e', e)$, $x'' \leftarrow \mathcal{H}'(e, e'')$ and $x \leftarrow \mathcal{H}'(e', e'')$ are computed. Let v_1, v_2, v_h be the path above them. Let $\phi(v), wt(v)$ denotes accumulator and witness stored in v resp. Now, for *op* = add, we do the following.

1. At level h, x', x'' are inserted and x is removed. The client can calculate and upload their witnesses as $wt(x') \leftarrow \{\phi(v_1)\}^{\frac{(\mathcal{H}(x')+s)}{(\mathcal{H}(x)+s)}}$ and $wt(x'') \leftarrow \{\phi(v_1)\}^{\frac{(\mathcal{H}(x'')+s)}{(\mathcal{H}(x)+s)}}$

2. For v_1, client computes $\phi_1(v_1) \leftarrow \{\phi(v_1)\}^{\mathcal{H}(x')+s}$, $\phi_2(v_1) \leftarrow \{\phi_1(v_1)\}^{\mathcal{H}(x'')+s}$ and $\phi_0(v_1) \leftarrow \{\phi_2(v_1)\}^{\frac{1}{\mathcal{H}(x)+s}}$.
   It computes $\phi_1(v_i) \leftarrow (\phi(v_i))^{(\mathcal{H}(\phi_0(v_{i-1}))+s)}$ and $\phi_0(v_i) \leftarrow (\phi_1(v_i))^{\frac{1}{(\mathcal{H}(\phi(v_i-1))+s)}}$, for other v_is. Thus, the new accumulator values along the path are $\phi_0(v_1), \phi_0(v_2), \ldots, \phi_0(v_h)$.

3. For each child u of v_1, cloud server computes updated witness $wt_0(u)$ without using s directly as follow.

   (a) Compute $wt_1(u) \leftarrow \phi(v_1).(wt(u))^{(\mathcal{H}(x')-\mathcal{H}(u))}$

   (b) Compute $wt_2(u) \leftarrow \phi_1(v_1).(wt_1(u))^{(\mathcal{H}(x'')-\mathcal{H}(u))}$

(c) compute $wt_0(u) \leftarrow \left(\frac{wt_2(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(x)-\mathcal{H}(u)}}$

4. Finally for any other child u of v_i new witnesses computed by the cloud server are $wt_1(u) \leftarrow \phi(v_1).(wt(u))^{\mathcal{H}(\phi_0(v_i))-\mathcal{H}(u)}$ and $wt_0(u) \leftarrow \left(\frac{wt_1(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(\phi(v_i))-\mathcal{H}(u)}}$

5. The client keeps $d' = \phi(v_h)$ as the new digest of the root

The client needs to keep the new digest only. Verification of the update is not required. If the cloud server changes something, no search result will be verified correctly.

If $op =$ delete, the tree can be updated in a similar way. The only changes in the algorithm are membership witnesses update of the leaf nodes (of the same bucket) and updating $\phi(v_1)$. During deletion, at Level-h, x is inserted and x', x'' are removed and the tree is updated accordingly as follows.

1. For v_1, client computes updated witness $\phi_0(v)$ of $\phi(v)$ as

$$\begin{aligned}\phi_1(v_1) &\leftarrow \{\phi(v_1)\}^{(\mathcal{H}(x)+s)},\\ \phi_2(v_1) &\leftarrow \{\phi_1(v_1)\}^{\frac{1}{\mathcal{H}(x)+s}}, \text{ and}\\ \phi_0(v_1) &\leftarrow \{\phi_2(v_1)\}^{\frac{1}{\mathcal{H}(x'')+s}}.\end{aligned}$$

   For $1 < i <= h$, similarly as in delete, it computes the new accumulator values $\phi_0(v_1), \phi_0(v_2), \ldots, \phi_0(v_h)$.

2. For each child u of v_1, cloud server computes updated witness $wt_0(u)$ without using s directly as follow.

   (a) Compute $wt_1(u) \leftarrow \phi(v_1).(wt(u))^{(\mathcal{H}(x)-\mathcal{H}(u))}$

   (b) Compute $wt_2(u) \leftarrow \left(\frac{wt_1(u)}{\phi_2(v_1)}\right)^{\frac{1}{\mathcal{H}(x')-\mathcal{H}(u)}}$

   (c) compute $wt_0(u) \leftarrow \left(\frac{wt_2(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(x'')-\mathcal{H}(u)}}$

3. Additionally, at Level-h, x is inserted and x', x'' are removed. Client computes the witnesses $wt(x) \leftarrow \{\phi(v_1)\}^{\frac{1}{(\mathcal{H}(x')+s)(\mathcal{H}(x'')+s)}}$.

4. Finally, for any other child u of v_i ($i > 1$) new witness $wt_0(u)$ computed similarly as $wt_0(u) \leftarrow \left(\frac{wt_1(u)}{\phi_0(v_1)}\right)^{\frac{1}{\mathcal{H}(\phi(v_i))-\mathcal{H}(u)}}$ where

$$wt_1(u) = \phi(v_1).(wt(u))^{(\mathcal{H}(\phi_0(v_i))-\mathcal{H}(u))}$$

5. The client keeps $d_0 = \phi(v_h)$ as the new digest of the root

6.2.3 Advantages of DIA tree

We have seen how the system works in a DIA tree, using on interval-approach. For a given set S, in both of [76] and our proposed DIA tree construction, the set is stored separately. In [76], leaf nodes stores $\mathcal{H}(x), \forall x \in S$ where $\mathcal{H}$ maps each element in a bilinear group G. However, in our case, we store maps of the open intervals as $\mathcal{H}((x, y)); x, y \in S$. It allows us to design the accumulation tree having both membership and non-membership proofs in a single structure.

They used a used an interval-based approach for non-membership proof only. They store ....

Papamanthou et al. [76] used an interval-based approach for non-membership proof only. They have to maintain two trees, one for membership proof and another for non-membership proof where the second one works only when the given set S is static. Secondly, they store the given set S, and an accumulation tree corresponding to the open intervals. However, *there is no formal description of how it works*.

Our proposed accumulation tree, DIA tree, gives proof of membership as well as non-membership even when the set S is dynamic. Moreover, it uses a single tree, resulting in reduction of cloud storage. We achieve those at the cost of computation. In DIA tree, the update time is thrice, and the search time is at most twice than that in [76]. However, the required time is asymptotically the same for both. We give computational complexity of DIA tree in Section 6.2.4.

There is another basic difference between the two constructions. In [76], the computation of witnesses is done by the cloud server when verification is required. This is useful when the frequency of the search is very low. However, if the search is frequent, we have to return only proofs fast. So, in the DIA tree we pre-compute all witnesses which enable the frequent search. For the same, the client has to update the $O(mh)$ witnesses during each interval update.

Moreover, we can see that the sorted Merkle tree can solve the problem of both membership and non-membership proof with very efficiently. However, one downside of sorted Merkle hash trees is that even if a single element in the data set S is changed, that element may need to move

to a different leaf, and the entire hash tree will need to be recomputed from scratch. This can take $O(|S|)$ hash computations. Dia tree not only provides all the functionalities of Merkle tree but also supports an efficient update, requiring at most $O(log|S|)$ calculations to update an element.

6.2.4 Storage and computation complexity of DIA tree

Let h be the height of the tree. Since, the leaves store elements in G corresponding to the intervals, parents of the leaves store different numbers of elements. However, the tree is a complete m-ary tree from Level-0 to Level-$(h-1)$, i.e., without leaves.

Storage Cost: Since, the tree is an m-ary tree without the leaves, it can hold upto $b = m^{(h-1)}$ elements in Level-$(h-1)$ and each node at Level-$(h-1)$ can hold at most $\frac{2^\lambda}{m^{(h-1)}}$ elements. However, there may be some nodes at Level-$(h-1)$ that may not contain only one element. If the size of the set is n, then the number of leaves is $n + 1 + m^{(h-1)}$. Now, each node stores an accumulator of its children and a witness of its parent. The root only keeps an accumulator and every leaf keeps an element in G corresponding to its interval. Number of internal nodes, from root to Level-$(h-1)$ is $m^{(h-1)} + m^{(h-2)} + \ldots + 1 = (m^{(h)} - 1)/(m-1)$. Thus, the number of elements the DIA tree store is $2(\frac{m^h-1}{m-1} + m^{(h-1)} + 1 + n) - 1$. This is stored at the cloud-side. The client keeps only the root and the secret key.

Building Cost: The numbers of accumulators at internal nodes is $\frac{m^h-1}{m-1}$. Among them, $\frac{m^{(h-1)}-1}{m-1}$ accumulators, in the Level above $h-1$, are for set of size m each. The accumulators at Level-$(h-1)$ are for the set of average size $\frac{n-1}{m^h}$. Besides, the number of witnesses to be generated is $(\frac{m^h-1}{m-1} + m^{(h-1)} + 1) + (n-1) = \frac{m^h-1}{m-1} + m^{(h-1)} + n$ The above cost is a one time client-side cost.

Search Cost: During a search, the cloud has to return the accumulators and witnesses in the paths corresponding to the given intervals. The cloud can retrieve them $O(2(h+1))$ or $O(h+1)$ time depending on whether the search element exists or not. Thereafter, the cloud returns $4(h+1)$ group elements if the searched element exists, else returns $2(h+1)$ group elements.

Verification Cost: To verify the result, the verifier needs to compute $4(h+1)$ and $2h$ powers in G. The cost is half if the searched element does not exist.

Update Cost: During an update, an interval, the client retrieves all nodes in the path corresponding to the interval and all witnesses that are affected due to this change. The client only retrieves and updates $2(m(h-1)+1)$ accumulators and sends them back to the cloud. The cloud stores them and updates the witnesses to their children. The number of such witnesses is $m(h-1)$ for the nodes

above Level-h. Other than that, there can have $2^{\lambda}/b$ witnesses at the bottom at most whereas the number is $|S|/b$ on average . The cost is double when the searched interval exists in the database.

6.2.5 Security of a DIA tree

We see that $\Phi(v)$ gives an accumulation tree corresponding to the subset of the set S rooted at v. Thus, $\Phi(v)$ is also called the *bilinear digest* of the tree rooted at v.

Theorem 6.1 (Acc tree security)**.** *Given a security parameter λ and a set $U = \{x_1, x_2, \ldots, x_n\}$, $x_i \in \mathsf{G}$, let DT be the accumulation tree constructed with $\mathtt{AC.Gen}()$ as above. Under the q-strong Diffie-Hellman assumption, the probability that a PPT adversary $\mathcal{A}$, knowing only the bilinear pairings parameters $(p, \mathsf{G}, \mathsf{G}, \hat{\mathsf{e}}, \mathsf{g})$ and the elements $\{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$, of G, for some randomly chosen s from $\mathsf{Z}_{\mathsf{p}}^*$ and $n \leq q$, can find another set V, with elements from G, such that $V \neq U$ and $\Phi(V) = \Phi(U)$ is $neg(k)$.*

Proof. Follows from the proof of Papamanthou et al. [76]. □

Theorem 6.2 (Security of our construction)**.** *Given a security parameter λ and a set $S = \{e_1, e_2, \ldots, e_n\}$, where $e_i \in \{0,1\}^*$, let $\mathtt{DIAT}$ be the accumulation tree constructed as above. Under the q-strong Diffie-Hellman assumption, the probability that a PPT adversary $\mathcal{A}$, knowing only the bilinear pairings parameters $(p, \mathsf{G}, \mathsf{G}, \hat{\mathsf{e}}, \mathsf{g})$ and the elements $\{g, g^{s^1}, g^{s^2}, \ldots, g^{s^q}\}$ of G, for some randomly chosen s from $\{0,1\}^*$ and $n \leq q$, can find another set S', with elements from G, such that $S' \neq S$ and $\Phi(S') = \Phi(S)$ is $neg(\lambda)$.*

Proof. Here, we use Theorem 6.1 with reduction method. We show that if Theorem 6.2 is false, then so is Theorem 6.1. But, since Theorem 6.1 is true, it implies Theorem 6.2 is true.

The main difference between our $\mathtt{DIAT}$ and Papamanthou et al. [76] is that our scheme supports efficient updates.

Since $\mathcal{H}$ is public, if $S = \{e_1, e_2, \ldots, e_n\}$ is given then so is $\bar{S} = \{x_1, x_2, \ldots, x_n\}$. Let us consider Theorem 6.2 does not hold, then there exists a PPT algorithm $\mathcal{A}$, which finds another set $S' = \{e'_1, e'_2, \ldots, e'_{n'}\}$ such that $\Phi(S') = \Phi(S)$ with probability $\geq neg(\lambda)$.

Let $U = \bar{S}$ and $V = \bar{S}'$ where $\bar{S}' = \{\mathcal{H}(I'_0), \mathcal{H}(I'_1), \ldots, \mathcal{H}(I'_{n'})\}$ and $I'_i = (e'_i, e'_{i+1}), \forall i$. Thus, given U, we have found a PPT adversary $\mathcal{A}$ that finds another set V with probability $\geq neg(\lambda)$. This contradicts Theorem 6.1. Thus our assumption that Theorem 6.2 does not hold is false. Hence Theorem 6.2 is true.

□

6.3 Our proposed VDCSE scheme

In this section, we propose our scheme which is forward private and verifiable. Our scheme does not use any extra storage for verification. Before discussing our scheme, we show that verifiability with $O(|W|)$ client storage is very easy for any single, conjunctive as well as Boolean keyword search schemes

DSE with $O(|W|)$ extra storage for verifiability: For *single keyword searches* [82] shows that when there is client storage of $O(|W|)$, verifiability can be achieved with any hash function for static data. Whereas, the same can be achieved with multiset hash (see Definition 2.9) when the data is dynamic. These schemes are for single keyword search only. Besides, for dynamic data, when forward privacy is concerned, the solution [107, 82] shows how forward privacy can be achieved without extra client storage and still with keeping them verifiable. We see that if for every keyword $w \in W$, the client is able to store a digest of the set of identifiers $DB(w)$, then any multiset hash H solves the problem of verifiability for a single keyword search. The client can compute an aggregated hash using multiset hashing, which can be updated with every update done by the client. The client can recompute the aggregated hash when receives search results for the keyword and can match with the stored one. In such a scenario, since all computations are done by the client and nothing is outsourced to the cloud, there is no forgery. Thus, with $O(|W|)$ client storage, the client is able to verify the result, using multiset hashing, in any single keyword search DSE scheme, without affecting the forward or backward privacy.

A *conjunctive forward private keyword* search scheme can be either static or dynamic. A dynamic conjunctive search may have forward privacy. With non-trivial solution[1], [45] deals with verifiability when data is static and [91, 107] deal when they are dynamic. However, when forward privacy is concerned, the above solutions are not applicable. Also, they used at least $O(|W|)$ client storage as well. In a conjunctive dynamic SE scheme, if the client is able to store the accumulator corresponding to each keyword $w \in W$, then the client is able to verify the received result. It can verify whether all resulted identifiers are present in a keyword or not. Since this requires extra computation the client can outsource this computation to a proxy server too. Thus, the extra $O(|W|)$ client storage makes the scheme easier to verify the search result for any conjunctive dynamic SE scheme without effecting its forward or backward privacy if there is any.

DSE without $O(|W|)$ extra storage: We see that $O(|W|)$ client storage can make any conjunc-

[1] A trivial solution is downloading search results for all keywords present in a conjunctive query and taking the intersection of them at client-side

tive as well as single and Boolean keyword search scheme verifiable. So, we are interested in the verifiability without this extra client storage. In this section, we propose a forward private conjunctive DSE scheme with verifiability that does not use any extra client storage for verifiability.

Trivially, if the client issues a single keyword search token for each keyword in the conjunctive query and cloud server returns the search result for each of them, then it can compute the intersections of them to get the final result. This can also be done using [82] without extra client storage. However, the trivial approach has two issues. Firstly, it leaks the complete result for each keyword instead of the required. Secondly, searching for identifiers containing each keyword requires extra computation power. Thus, it is inefficient for a conjunctive search.

Difficulty in extending existing schemes to a VDCSE scheme: Previous conjunctive DSE schemes are either verifiable without forward privacy ([96]) or are forward private without verifiability ([99]). There are other conjunctive schemes which are neither forward private nor conjunctive. An obvious question is whether existing conjunctive DSE without forward privacy [99] or without verifiability [96] can be extended with having both.

The key point of [99] is that modification of documents is not allowed here and the files are always unchanged. So, if we keep the accumulators at the leaf node of VBTree corresponding to every document, the accumulators will be always unchanged. However, the solution is not complete. If we keep the accumulators in the leaf then, for membership or non-membership proof, it must reveal what are the elements in the set. Thus the tree structure will be revealed and the scheme cannot be forward private anymore. The cloud sends which tuple is not present in a node in the path. So we have to keep the accumulators in the nodes of the tree. However, if some new file is added then the complete path of the file may be revealed to add new elements in the nodes. *Thus, it is hard to extend [99] to be verifiable without extra client storage.*

Besides, [96] is for static data. So forward privacy is not applicable to it. If we extend the scheme to be dynamic then we can only try to make it forward private. We can see that it keeps the accumulator corresponding to every keyword on the cloud-side. When an update happens, the cloud has to update the accumulators too, and updating it reveals whether the keyword was searched previously. So, its extension to forward private is not possible in this way.

6.3.1 Overview of our proposed scheme Blasu

In this section, we present a generic forward private conjunctive DSE scheme with verifiability that makes any forward private single keyword search scheme to conjunctive one. However, we want

to reduce this extra client storage for verifiability. Our proposed forward private conjunctive DSE scheme with verifiability does not use any extra client storage for verifiability. Here, we give a short overview of our scheme Blasu.

In most of the conjunctive schemes, including ours, the least-frequency method is considered (exception [99]). In this method, the least frequent keyword is taken and its result is found. Then for each resulting document, the presence of all other keywords is checked. For example, given a query $\hat{w} = \{w_1, w_2, \ldots, w_k\}$, let $R_{w_1} = \{id_1^{w_1}, id_2^{w_1}, \ldots, id_{n_{w_1}}^{w_1}\}$ be the single keyword query result for the lowest frequent keyword w_1. The frequency of a keyword is the number of documents that contain it. The cloud server computes R_{w_1} and checks if id_i^1, for $1 \leq i \leq n_{w_1}$, contains all the keywords in $\hat{w} \setminus \{w_1\}$ and includes it in the search result $R_{\hat{w}}$, in the case does.

Our proposed scheme Blasu uses a forward private DSE scheme Σ as a black box.At the time of building two data structures, a hash table, and a DIA tree are built in addition to the encrypted index. For each keyword-document pair, a unique element is created. The element stores a signature generated using the corresponding keyword and the document identifier. It is kept in the hash table that gives it efficient access. After all the elements are generated, a DIA tree is built on them.

To search, we use the least-frequency approach. The client first generates a search token and sends it to the cloud. The cloud performs a search using Σ for a minimal frequent keyword. Then, for each document identifier in the result, the cloud checks its existence of other keywords. It returns the documents, each of which contains all searched keywords together with proof of its correct execution. The elements and DIA tree help the cloud to return the proof. To update a new keyword-document pair, the client generates the corresponding element and sends it to the cloud, which then updates both the table and the tree accordingly.

6.3.2 Technical details

There are three phases in our proposed VDCSE scheme Blasu, denoted by Ψ, which is a tuple (`VCKeyGen`, `VCBuildIndex`, `VCSrchTknGen`, `VCSearchCD`, `VCSearchCT`, `VCUpdtTknGen`, `VCUpdate`)– initialization, search and update. In the first phase, the client generates keys and builds encrypted search index. Then it generates the DIA tree. In the search phase, the client generates search token and sends it to the cloud. Then the cloud performs search over the data and generates proof and sends the result to the client. The client generates some proof from the result it gets so that anyone can verify the proofs. The interaction between the entities, during those phases, are shown in Fig 6-5. The phases are described as follows.

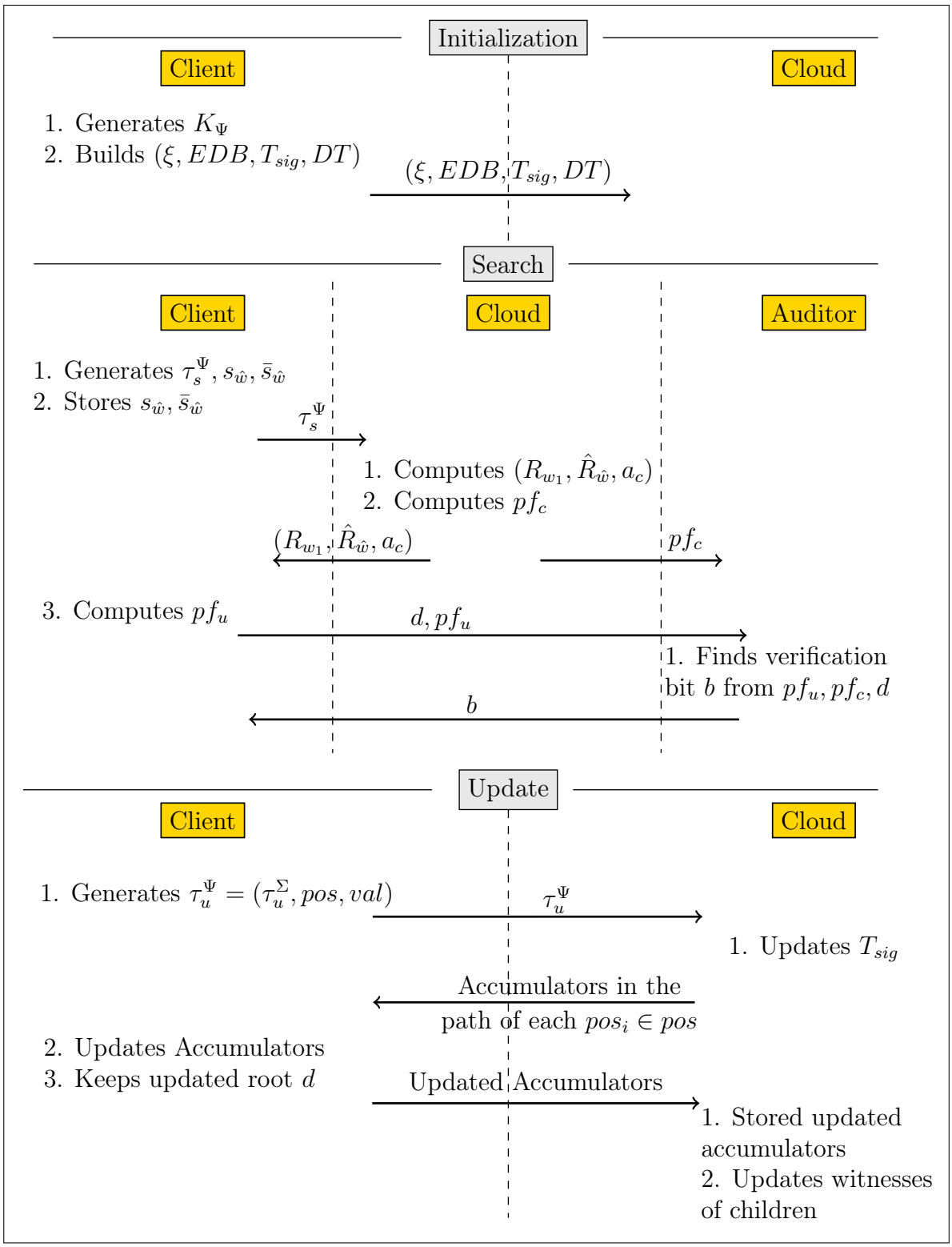


Figure 6-5: Interaction between entities in different phases

Initialization: It is divided into two parts– key generation and building an encrypted search index, given as below.

Key Generation: is given Algo. 16. Let $\mathcal{E} = (\texttt{Enc}, \texttt{Dec})$ be a CPA-secure symmetric encryption scheme with key-space $\{0,1\}^\lambda$. Given some security parameter λ, the key K_Σ is generated for Σ. Moreover, three λ-bit stings K_s, K_t and $K_{\bar{s}}$ are picked at random to use them as secret.

Algorithm 16: $\Psi.\texttt{VCKeyGen}(1^\lambda)$

$K_\Sigma \leftarrow \Sigma.\texttt{KeyGen}(1^\lambda)$; $(sk, pk) \leftarrow \mathcal{S}.\texttt{KeyGen}(1^\lambda)$;
$K_s, K_t, K_{\bar{s}} \leftarrow \{0,1\}^\lambda$; $s \leftarrow \{0,1\}^\lambda$ /*for DIAT*/;
return $K_\Psi = (K_s, K_t, K_{\bar{s}}, sk, pk, K_\Sigma, s)$;

For each keyword, K_s is used as a key, together with the keyword, for generating a seed. This seed is used to generate a sequence of random numbers. Similarly, for each keyword, K_t helps to generate a tag that helps to find some random positions in a table (T_{sig}). Whereas $K_{\bar{s}}$ generates a unique key for the symmetric encryption scheme $\mathcal{E}$. A BLS signature scheme $\mathcal{S} = (\texttt{KeyGen}, \texttt{Sign}, \texttt{Verify})$ is generated together with a tuple $tup = (p, \mathsf{G}, \mathsf{G}, \hat{\mathsf{e}}, \mathsf{g})$ and a key pair (sk, pk). A λ-bit secret key s is chosen for DIA tree DT. Finally, $K_\Psi = (K_s, K_{\bar{s}}, K_t, sk, pk, K_\Sigma)$ is returned

Encrypted Index Building: is given in Algo 17. Instead of $DB(w)$, for each $w \in W$, we consider $DB(w) \cup \{id_0^w\}$, where $\{id_0^w\}$ is a random unassigned identifier. Doing so prevents the cloud server to return an empty set of identifiers. Whenever the cloud returns the actual file it neglects the first identifier. Without loss of generality, we take $DB(w) = DB(w) \cup \{id_0^w\}$ and $\mathcal{DB} = \{DB(w) \cup \{id_0^w\} : w \in W\}$. In rest of the chapter, we consider the same.

To generate an encrypted search index, the client takes an empty hash table T_{sig} where it keeps a key-value pair $(pos_i^w, (\sigma_i^w, v_i^w))$ for each keyword-doc pair (w, id_i^w). The key pos_i^w indicates the position in the table where value (σ_i^w, v_i^w) keeps two things– a signature σ_i^w for the pair and encrypted file-sequence-number v_i^w for the keywords. A symmetric key encryption scheme Enc can be taken to get v_i^w. Finally, the client builds a DIA tree DT for the set P of all such positions. The tree DT is constructed with $\texttt{DIAT.BuildTree}(tup, s, P)$ as described in Section 6.2. The root of the tree is kept at client-side. Moreover, the documents are kept encrypted. The encrypted index ξ for them is generated using Σ. Here $\gamma = \{\xi, T_{Sig}\}$ and $\mathcal{I} = DT$.

Search Phase: The search phase consists of three steps. At first, the client generates a search token to search for a set of keywords and sends it to the cloud server. Then, the cloud server performs a search on the encrypted database and generates a proof of the search result. The client

Algorithm 17: $\Psi.\texttt{VCBuild}(\mathcal{DB}, K_\Psi)$

$T_{sig} \leftarrow$ An empty list of size $|\mathcal{W}|$;
for $w \in \mathcal{W}$ **do**
  $s_w \leftarrow F(K_s, w)$; $tag_w \leftarrow F(K_t, w)$; $\bar{s}_w \leftarrow F(K_{\bar{s}}, w)$;
  for $i = 0$ ***to*** $c_w (= |DB(w)|)$ **do**
    $r_i^w \leftarrow R(s_w||i)$;
    $m_i^w \leftarrow r_i^w.id_i^w \pmod q$; $pos_i^w \leftarrow F(tag_w, id_i^w||i)$;
    $\sigma_i^w \leftarrow \mathcal{S}.\texttt{Sign}(sk, m_i^w)$; $v_i^w \leftarrow \mathcal{E}.\texttt{Enc}(\bar{s}_w, i)$;
    $T_{sig}[pos_i^w] \leftarrow (\sigma_i^w, v_i^w)$;
  end
end
$P = \{pos_i^w : w \in \mathcal{W}$ and $i = 0, 1, \ldots, n_w\}$;
$(DT, d) \leftarrow \texttt{DIAT.BuildTree}(tup, s, P)$;
$\mathcal{DB} = \{DB(w) : w \in W\}$;
$(\xi, EDB) \leftarrow \Sigma.Build(\mathcal{DB}, K_\Sigma)$;
Client keeps the root digest d;
return (ξ, EDB, T_{sig}, DT) to cloud;

also generates a proof of the received result and gives it to the auditor. Finally, an auditor verifies the result and the proofs.

Search token generation: Given a query $\hat{w} = \{w_1, w_2, \ldots, w_{|\hat{w}|}\}$, the client first generates search token $\tau_{w_1}^\Sigma$ for w_1 (the lowest frequent keyword), according to the base searchable encryption scheme Σ (Algo. 18). This helps to find file identifiers that contain w_1. Then, it generates corresponding set of tags $\{tag_{w_1}, tag_{w_2}, \ldots, tag_{w_{|\hat{w}|}}\}$ for all keywords. These tags help to find whether the keyword-document pairs exist without revealing the actual keyword. Additionally, $s_{\hat{w}} = \{s_{w_1}, s_{w_2}, \ldots, s_{w_{|\hat{w}|}}\}$ and $\bar{s}_{\hat{w}} = \{\bar{s}_{w_1}, \bar{s}_{w_2}, \ldots, \bar{s}_{w_{|\hat{w}|}}\}$ are generated by the client. Finally, $\tau_s^\Psi = (\tau_{w_1}^\Sigma, tag_{w_1}, tag_{w_2}, \ldots, tag_{w_{|\hat{w}|}})$ is issued as a search token for the cloud and $s_{\hat{w}}, \bar{s}_{\hat{w}}$ are stored at client-side.

Search and proof generation: After receiving the search token τ_s^Ψ from the client, at first, the cloud finds single keyword search result R_{w_1}=$\{id_0^{w_1}, id_1^{w_1}, \ldots, id_{n_{w_1}}^{w_1}\}$ for the keyword w_1 using Σ. Then from R_{w_1} and the tag tag_{w_1}, it finds the position of the keyword-file pairs corresponding to w_1 and retrieves signatures of them from the table T_{sig}. After that, it multiplies them as an aggregate signature for w_1 which is treated proof $pf_c^{(0)}$ for w_1.

Then, for each file in R_{w_1}, it checks whether other keywords are present in the file by verifying the existence of the keyword-file pairs. To verify them, corresponding positions are regenerated and checked whether the table T_{sig} contains them. If for some jth file id, the position does not exist,

Algorithm 18: $\Psi.\texttt{VCSrchTknGen}(\hat{w}, K_\Psi)$

$\{w_1, w_2, \ldots, w_{|\hat{w}|}\} \leftarrow \hat{w}$, where w_1 is minimal frequent ;
$(K_s, K_t, K_{\bar{s}}, sk, pk, K_\Sigma, s) \leftarrow K_\Psi$;
$\tau^{\Sigma}_{w_1} \leftarrow \Sigma.\texttt{SearchToken}(w_1, K_\Sigma)$;
for $i = 1$ ***to*** $i = |\hat{w}|$ **do**
  $tag_{w_i} \leftarrow F(K_t, w_i)$; $s_{w_i} \leftarrow F(K_s, w_i)$; $\bar{s}_{w_i} \leftarrow F(K_{\bar{s}}, w_i)$;
end
$\tau^{\Psi}_s \leftarrow (\tau^{\Sigma}_{w_1}, tag_{w_1}, \ldots, tag_{w_{|\hat{w}|}})$;
$s_{\hat{w}} \leftarrow (s_{w_1}, s_{w_2}, \ldots, s_{w_{|\hat{w}|}})$; $\bar{s}_{\hat{w}} \leftarrow (\bar{s}_{w_1}, \bar{s}_{w_2}, \ldots, \bar{s}_{w_{|\hat{w}|}})$;
return τ^{Ψ}_s for cloud and $(s_{\hat{w}}, \bar{s}_{\hat{w}})$ only for client;

the cloud computes non-membership proof $pf_c^{(i)}$ for that positions. If all keywords are contained in jth file, then the product of their signatures is returned as proof corresponding to the keyword, and the set $a_c^{(j)}$ of v_i^js are returned to the client. $\hat{R}_{\hat{w}}$ keeps the identifiers that contain all keywords.

Thus, for each file in R_{w_1}, if it is in $\hat{R}_{\hat{w}}$, then the cloud returns the product of the signatures corresponding to the keyword file pairs and the set of v_i^js. In case a file in R_{w_1} is not present in $\hat{R}_{\hat{w}}$, it returns a non-membership proof for the position corresponding to a non-existing keyword-file pair. Finally, the cloud server returns its part of the proof pf_c and $(\hat{R}_{\hat{w}}, X_{\hat{w}})$ to the client where $X_{\hat{w}} = (R_{w_1}, a_c)$ and a_c is the auxiliary information from cloud.

After receiving$(\hat{R}_{\hat{w}}, X_{\hat{w}} = (R_{w_1}, a_c))$, the client generates its part of the proof pf_u. For w_1, it regenerates all the random numbers $m_i^{w_1}$ for each of the files in R_{w_1}. Then it generates the product of them as $m_0 = \sum_{i=0}^{n_{w_1}} m_i^{w_1} \pmod p$ (see step 3 to step 10 in in Algo. 20).

For each file $id \in R_{w_1} \setminus \{w_1\}$, if $id \in \hat{R}_{\hat{w}}$, the client decrypts the encrypted numbers v_i^js, generates random numbers corresponding to each keyword and calculates the product m_i of them as $pf_u^{(i)}$. This acts as membership proof of all the keywords in the file. So, we do not have to generate a separate proof for all keyword-file pairs. In case, $id \notin \hat{R}_{\hat{w}}$, the client keeps $pf_u^{(i)}$ as null. This is because the cloud already keeps non-membership proof for them.

The auditor (or any third party) verifies the search result by taking pf_c from the cloud, and pf_u, $\hat{R}_{\hat{w}}$ and d form the client. The algorithm is given in Algo. 20.

Verification: The auditor verifies for w_1 as well as for each files in R_{w_1}. There are two cases in verification. For the identifiers $\in \hat{R}_{\hat{w}}$, containing all keywords, it verifies $\mathcal{S}.\texttt{Verify}(pk, pf_u[k][1], pf_c[k])$. For the identifiers $\notin \hat{R}_{\hat{w}}$ that does not contains some keyword, non-membership proof, for corresponding pos, is verified with $\texttt{DIAT.VerifySearch}$. auditor returns *accept* only when all get success (see Algo. 21).

Algorithm 19: $\Psi.\texttt{VCSearchCD}(\gamma, \tau_s^{\Psi})$

Cloud Receives τ_s^{Ψ} from client ;
$(\tau_{w_1}^{\Sigma}, tag_{w_1}, tag_{w_2}, \ldots, tag_{w_{|\hat{w}|}}) \leftarrow \tau_s^{\Psi}$;
$R_{w_1} \leftarrow \Sigma.\texttt{Search}(\xi, \tau_{w_1}^{\Sigma})$;
$\{id_0^{w_1}, id_1^{w_1}, \ldots, id_{n_{w_1}}^{w_1}\} = R_{w_1}$;
for $i = 0$ ***to*** n_{w_1} **do**
    $pos_i^{w_1} \leftarrow F(tag_{w_1}, id_i^{w_1}||i)$;
    $\sigma_i' \leftarrow T_{sig}[pos_i^{w_1}][0]$; ;
end
$pf_c^{(0)} = \sigma' \leftarrow \prod_{i=0}^{n_w} \sigma_i'$; $\hat{R}_{\hat{w}} \leftarrow \Phi$;
if $|\hat{w}| = 1$ **return** $(R_{w_1}, pf_c^{(1)})$
for $j = 1$ ***to*** n_{w_1} **do**
    $flag = 0$;
    for $i = 2$ ***to*** $|\hat{w}|$ **do**
        $pos_j^{w_i} \leftarrow F(tag_{w_i}, id_j^{w_1})$;
        if $[T_{sig}[pos_j^{w_i}]] = \perp$ **then**
            $pf_c^{(j)} \leftarrow \texttt{DIAT.Search}(DT, pos_j^{w_i})$;
            $a_c^{(j)} \leftarrow pos_j^{w_i}$; $flag = 1$;
            break;
        end
        $(\sigma_j^i, v_j^i) \leftarrow T_{sig}[pos_j^{w_i}]$;
    end
    if $flag = 0$ **then**
        $pf_c^{(j)} \leftarrow \prod_{i=2}^{|\hat{w}|} \sigma_j^i$; $a_c^{(j)} \leftarrow (v_j^2, \ldots, v_j^{|\hat{w}|})$;
        $\hat{R}_{\hat{w}} \leftarrow \hat{R}_{\hat{w}} \cup \{id_j^{w_1}\}$;
    end
end
$pf_c = (pf_c^{(0)}, pf_c^{(1)}, \ldots, pf_c^{(n_{w_1})})$;
$a_c = (a_c^{(1)}, a_c^{(2)}, \ldots, a_c^{(n_{w_1})})$; $X_{\hat{w}} = (R_{w_1}, a_c)$;
return pf_c and $(\hat{R}_{\hat{w}}, X_{\hat{w}})$;

Algorithm 20: $\Psi.\texttt{VCSearchCT}(\gamma, \tau_s^{\Psi})$

Client Receives $(\hat{R}_{\hat{w}}, X_{\hat{w}} = (R_{w_1}, a_c))$;
$(a_c^{(1)}, a_c^{(2)}, \ldots, a_c^{(n_{w_1})}) \leftarrow a_c$;
$\{id_0^{w_1}, id_1^{w_1}, \ldots, id_{n'_{w_1}}^{w_1}\} \leftarrow R_{w_1}$; $n_{w_1} \leftarrow C[w_1]$;
if $n_{w_1} \neq n'_{w_1}$ **then return** reject;
$\{s_{w_1}, s_{w_2}, \ldots, s_{w_{|\hat{w}|}}\} \leftarrow s_{\hat{w}}$ (see Algo. 18);
for $i = 0$ ***to*** n_{w_1} **do**
  $r_i^{w_1} \leftarrow R(s_{w_1}||i)$;
  $m_i^{w_1} \leftarrow id_i^{w_1}.r_i^{w_1} \pmod p$;
end
$pf_u^{(0)} = m_0 = \sum_{i=0}^{n_{w_1}} m_i^{w_1} \pmod p$;
for $j = 1$ ***to*** n_{w_1} **do**
  if $id_j^{w_1} \notin \hat{R}_{\hat{w}}$ **then** $pf_u^{(i)} = (0, a_c^{(j)})$;
  else
    $(v_j^2, v_j^3, \ldots, v_j^{|\hat{w}|}) \leftarrow a_c^{(j)}$;
    for $i = 2$ ***to*** $|\hat{w}|$ **do**
      $k_i \leftarrow \mathcal{E}.\texttt{Dec}(\bar{s}_{w_i}, v_j^i)$;
      $r_i^j \leftarrow R(s_{w_i}||k_i)$;
      $m_i^j \leftarrow \hat{R}_{\hat{w}}[i].r_i^j \pmod p$;
    end
    $m_j = \sum_{i=2}^{|\hat{w}|} m_i^j \pmod p$;
    $pf_u^{(j)} = (1, m_j)$
  end
end
return $pf_u = \{pf_u^{(0)}, pf_u^{(1)}, \ldots, pf_u^{(n_{w_1})}\}$;

Algo. 21: $\Psi.\texttt{Verify}(d, pf_u, pf_c, \hat{R}_{\hat{w}})$

Receives pf_u from client and pf_c from cloud;
for $k = 0$ **to** n_{w_1} **do**
  if $pf_u[k][0] = 0$ **then**
    $b_v = \texttt{DIAT.}\ \texttt{VerifySearch}(d, pf_c[k][0], pf_c[k][1], pf_u[k][1])$
  else
    $b_v \leftarrow \mathcal{S}.\texttt{Verify}(pk, pf_u[k][1], pf_c[k])$
  end
  if $b_v =$ *failure* **return** *reject*;
end
return *accept* ;

Updating the database: Given a new file f with a new identifier id, the client first generates an update token. From f, it extracts the set of keywords $\{w_1, w_2, \ldots, w_{n_{id}}\}$, where n_{id} is the number of keywords present in f. It computes update token τ_u^{Σ} of the file according to Σ. For each keyword-doc pair, during update, corresponding entries in T_{sig} and the DIA tree DT are updated. Since, the client stores the frequencies of the keywords as the state, it retrieves them to compute key-value pairs for the table T_{sig}.

For each keyword w_i, it generates tag tag_{w_i}, s_{w_i} and $\bar{s}_{w_i}$ with the secret key. Then it generates key-value pair (pos_i, val_i) for every keywords w_i as given in Algo. 23. Finally, it returns $\tau_u^{\Psi} = (\tau_u^{\Sigma}, pos, val)$ to the cloud.

Algo. 22: $\Psi.\texttt{VCUpdtTkn}(K_{\Psi}, st, f)$

$\{w_1, w_2, \ldots, w_{n_{id}}\} \in f$;
$(K_t, K_s, sk, pk, K_{\Sigma}) \leftarrow K_{\Psi}$;
$\tau_u^{\Sigma} \leftarrow \Sigma.\texttt{UpdateToken}(K_{\Sigma}, w_i, id)\ \forall i \in [n_{id}]$;
for $i = 1$ ***to*** n_{id} **do**
  $s_{w_i} \leftarrow F(K_s, w_i)$; $tag_{w_i} \leftarrow F(K_t, w_i)$;
  $\bar{s}_{w_i} \leftarrow F(K_{\bar{s}}, w_i)$; $n_{w_i} \leftarrow C[w_i]$;
  $r_i \leftarrow R(s_{w_i}||(n_{w_i}+1))$; $C[w_i] = C[w_i] + 1$;
  $m_i \leftarrow r_i.id \pmod p$; $v_i \leftarrow \mathcal{E}.\texttt{Enc}(\bar{s}_w, c_w + 1)$;
  $\sigma_i \leftarrow \mathcal{S}.\texttt{Sign}(sk, m_i^{w_i})$;
  $pos_i \leftarrow F(tag_{w_i}, id)$; $val_i = (\sigma_i, v_i)$
end
$pos \leftarrow \{pos_1, pos_2, \ldots, pos_{n_{id}}\}$;
$val \leftarrow \{val_1, val_2, \ldots, val_{n_{id}}\}$;
return $\tau_u^{\Psi} = (\tau_u^{\Sigma}, pos, val)$

During the update phase, the cloud updates the file f according to Σ. Then it inserts key-value pairs in the table T_{sig}. Finally, after updating them in the database, it updates DT for each pos_i and returns corresponding proof of update for each position.

Extra cost for verifiability: Building the index requires $O(N)$ key-value pairs computation and a DIA tree for a set of size N. During the search, the cloud server has to compute $O((|\hat{w}|+1).|R_{w_1}|)$ key-value pair, $O(|\hat{R}_{\hat{w}}|.|R_{w_1}|)$ multiplications in G. It also has to compute $O(|\hat{w}|.(|R_{w_1}| - |\hat{R}_{\hat{w}}|))$ key-value pairs together with proofs of their non-membership. To generate proof at the client-side, the client only generates random numbers and computes the product of them which makes them very efficient for lightweight clients.

Algo. 23: Ψ.VCUpdate(T_{tag}, γ, op, f)

$(\tau_u^{\Sigma}, pos, val) = \tau_u^{\Psi}$;
Σ.Update$(\xi, \tau_u^{\Sigma}, op)$;
$\{pos_1, pos_2, \ldots, pos_n\} \leftarrow pos$; $\{val_1, val_2, \ldots, val_n\} \leftarrow val$;
for $i = 1$**to** $i = n$ **do**
  if (op=add) **then** $T_{sig}[pos_i] \leftarrow val_i$;
  else remove $T_{sig}[pos_i]$;
  $inpt \leftarrow (DT, s, pos_i, op, d)$;
  $d' \leftarrow$ DIAT.Update$(inpt)$
end
Client keeps updated d';
return

6.3.3 Security of our proposed scheme

6.3.3.1 Confidentiality

We see that the DIA tree is just an additional data structure that is get searched (updated) when the key of a key-value pair is searched (updated). So, it does not give any extra information about the encrypted database. (At the time of simulation, the simulator can also keep a similar tree based on the simulated database. The simulator only gives existential proof. So, in our security proof, we have not taken the accumulator part.) Else, suppose we have stored with a list of entries, then we build a simulator corresponding to that simulated database. In either case, the simulator must be there and so is the DIA tree. Since a verification phase is there, we cannot return the random element in that case of the DIA tree. So, we can eliminate DT from leakage, but we should keep it with valid proof.

Leakage function: Let $\mathcal{L}_{\Sigma} = (\mathcal{L}_{\Sigma}^{bld}, \mathcal{L}_{\Sigma}^{srch}, \mathcal{L}_{\Sigma}^{updt})$ be the leakage function of Σ, then the leakage function $\mathcal{L}_{\Psi}$ = ($\mathcal{L}_{\Psi}^{bld}$, $\mathcal{L}_{\Psi}^{srch}$, $\mathcal{L}_{\Psi}^{updt}$) of Ψ is given as follows.

$$\begin{aligned}
\mathcal{L}_{\Psi}^{bld}(\mathcal{DB}) &= \{\mathcal{L}_{\Sigma}^{bld}(\mathcal{DB}), |T_{sig}|\} \\
\mathcal{L}_{\Psi}^{srch}(\hat{w}) &= \{\mathcal{L}_{\Sigma}^{srch}(w_1), \{(id_i^{w_1}, pos_i^{w_1}, \sigma_i^{w_1}) : i = 1, 2, \ldots, n_{w_1}\}, \\
&\quad \{(pos_j^i, \sigma_j^i) : \forall id_j \in R_{w_i}, w_i \in \hat{w}, i \neq 1\}\} \\
\mathcal{L}_{\Psi}^{updt}(w, id) &= \{id, \mathcal{L}_{\Sigma}^{updt}(w, id), pos^w, \sigma^w\}
\end{aligned}$$

Since we consider any forward private DSE scheme Σ which $\mathcal{L}_{\Sigma}$-secure against adaptive cho-

sen keyword attack, we have the following theorem.

Theorem 6.3. *Let* $\Sigma = (KeyGen,\ Build,\ SearchToken,\ Search,\ UpdateToken,\ Update)$ *be the forward private correct DSE scheme with leakage function* $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{bld}, \mathcal{L}_\Sigma^{srch}, \mathcal{L}_\Sigma^{updt})$. *If* Σ *is* $\mathcal{L}_\Sigma$*-secure against adaptive chosen keyword attack, under random oracle model, then for any adversary* $\mathcal{A}_\Sigma$*, there exists a simulator* $\mathcal{S}_\Sigma$ *which simulates* Σ.

The proof of the above theorem depends on the scheme Σ and can be seen in the corresponding paper (for example; [16]). However, assuming the theorem we will proof confidentiality of Ψ. We show that Ψ is $\mathcal{L}_\Psi$-secure against adaptive dynamic chosen-query attacks in the random oracle model. The proof of confidentiality is given as follows.

Theorem 6.4. *If* F *is a PRF,* R *is a PRG and* Σ *is* $\mathcal{L}_\Sigma$*-secure against adaptive dynamic chosen-query attacks in the random oracle model, then* Ψ *is* $\mathcal{L}_\Psi$*-secure against adaptive dynamic chosen-query attacks, under* q*-SDH assumption, in random oracle model.*

Proof. We give the proof of the above theorem, according to Definition 6.3. It is sufficient to show that, for any PPT adversary $\mathcal{A}_\Psi$, there exists a simulator $\mathcal{S}_\Psi$, for which, the output of $\mathbf{Real}^\Psi_{\mathcal{A}_\Psi}(\lambda)$ and $\mathbf{Ideal}^\Psi_{\mathcal{A}_\Psi,\mathcal{S}_\Psi}(\lambda)$ are computationally indistinguishable.

Let $\mathcal{A}_\Sigma$ be the part of $\mathcal{A}_\Psi$ for Σ, then by Theorem 6.3, there exists a simulator $\mathcal{S}_\Sigma$ that simulates Σ. Therefore, it is to remain to construct a simulator $\mathcal{S}_\Psi$ to simulated extra data structure T_{sig} and query tokens (both search and update). Then, $\mathcal{S}_\Psi$ simulates as follows.

Simulating F: Simulation of the PRF F is done using a table T_F in random oracle model. For a given pair (x, y) of elements in $\mathtt{G}$, if $T_F[(x, y)] = \perp$, i.e. the corresponding entry does not exists, a random entry is kept as $T_F[(x, y)] \leftarrow \{0, 1\}^\lambda$ and finally $T_F[(x, y)]$ is returned.

Simulating Build: Given the leakage $\mathcal{L}^{bld}_\Psi(\mathcal{DB}) = \{\mathcal{L}^{bld}_\Sigma(\mathcal{DB}), |T_{sig}|\}$, $\mathcal{S}$ simulates two data structure EDB and the table T_{sig}. The DIA tree always accumulates the keys of the key-value pairs of T_{sig}. Again, T_{sig} is simulated with a table $\widetilde{T}_{sig}$. While simulating, let $\mathcal{S}_\Sigma$ returns $\widetilde{DB}$ while simulating EDB.

To keep the tags, a table $\widetilde{T}_{tag}$ taken by $\mathtt{Sim}_\Psi$. It stores a random λ-bit string for every keyword w. It acts as random oracle and returns $\widetilde{tag}_w \leftarrow \widetilde{T}_{tag}[w]$. A table $\widetilde{T}'_{sig}$ is also kept by $\mathtt{Sim}_\Psi$ to indicate whether an entry $\widetilde{T}_{sig}$ is queried or not.

The building of the data structures are simulated as follows.

1. $\widetilde{T}_{sig} \leftarrow \Phi$ and $\widetilde{T}'_{sig} \leftarrow \Phi$
2. For $i = 1$ to $i = |T_{sig}|$ do

(a) $pos_i \xleftarrow{\$} \{0,1\}^\lambda$; $val_i \xleftarrow{\$} \{0,1\}^\lambda$

(b) $\widetilde{T}_{sig}[pos_i] \leftarrow val_i$; $\widetilde{T}'_{sig}[pos_i] \leftarrow 0$

3. $\widetilde{DB} \leftarrow \mathcal{S}_\Sigma(\mathcal{L}^{bld}_\Sigma(\mathcal{DB}))$
4. $tup = (p, \mathsf{G}, \mathsf{G}, \hat{e}, g) \leftarrow \mathtt{BMGen}(1^\lambda)$ is generated for $\widetilde{\mathtt{DT}}$.
5. $s \leftarrow \{0,1\}^\lambda$
6. $\widetilde{\mathtt{DIAT}}.build(tup, s, \{pos_i : i = 1, \ldots, |T_{sig}|\})$
7. return $(\widetilde{DB}, \widetilde{T}_{sig}, \widetilde{\mathtt{DT}})$ and keeps $(\widetilde{T}'_{sig}, s, p)$

Simulating search token: Let the search leakage $\mathcal{L}^{srch}_\Psi(\hat{w})$ is given.

A table T_F is taken to keep the positions for each keyword-file pair. Given a tuple $(\widetilde{tag}_w, id, i)$ it returns a position in the table. If the position is searched before, then it returns the previous one, else it allocate a new and return that. These table is kept at $\mathcal{S}_\Psi$. Let $\{w_1, w_2, \ldots, w_n\} \in \hat{w}$ where w_1 has least frequency. The complete simulation of search token is done by $\mathcal{S}_\Psi$ as follows.

1. Receives τ^Ψ_s from client ;
2. $\tau^\Sigma_{w_1} \leftarrow \mathtt{Sim}_\Sigma.\mathtt{SimSearch}(\mathcal{L}^\Sigma_{srch}(w_1))$;
3. For $i = 1$ **to** n'_{w_1}
   (a) $\widetilde{tag}_{w_i} \xleftarrow{o} \widetilde{T}_{tag}[w_i]$;
   (b) $pos^{w_1}_i \xleftarrow{o} T_F[(\widetilde{tag}_{w_1}, id^{w_1}_i || i)]$;
   (c) $\sigma'_i \xleftarrow{o} \widetilde{T}_{sig}[\widetilde{pos}^{w_1}_i]$;
4. $pf^{(1)}_c = \sigma' \leftarrow \prod_{i=1}^{n_w} \sigma'_i$;
5. For $j = 1$ **to** n'_{w_1}
   (a) For $i = 2$ **to** n_q
      i. $\widetilde{pos}^{w_i}_j \xleftarrow{o} T_F[(\widetilde{tag}_{w_i}, id^{w_1}_j)]$;
      ii. If $(T_{sig}[pos^{w_i}_j] == \perp)$
         - $\widetilde{T}_{sig}[\widetilde{pos}^{w_i}_j] \leftarrow \perp$
         - $pf^{(j)}_c \leftarrow \mathtt{DIAT.Search}(\widetilde{DT}, pos^{w_i}_j)$
         - $a^{(j)}_c \leftarrow null$;
         - continue for next i;
      iii. $(\sigma^i_j, v^i_j) \leftarrow \widetilde{T}_{sig}[pos^{w_i}_j]$;
   (b) $pf^{(j)}_c = \sigma_j \leftarrow \prod_{i=1}^{|\hat{w}|} \sigma^i_j$;
   (c) $a^{(j)}_c \leftarrow (v^1_j, v^2_j, \ldots, v^{|\hat{w}|}_j)$;
6. $pf_c = (pf^{(1)}_c, pf^{(2)}_c, \ldots, pf^{(n_{w_1})}_c)$;
7. $a_c = (a^{(1)}_c, a^{(2)}_c, \ldots, a^{(n_{w_1})}_c)$;
8. Return pf_c and $(R_{w_1}, \hat{R}_{\hat{w}}, a_c)$;
9. return $\widetilde{\tau}^\Psi_s = (\widetilde{\tau}_\Sigma, \widetilde{tag}_w)$

Here, oracle access is indicated by "$\xleftarrow{o}$", if the elements is not empty, then it is returned, else a random element is allocated and then returned.

Simulating Update token Leakage function to add a document f with identifier id containing keyword set $\{w_1, w_2, \ldots, w_{n_w}\}$ is given by

$\mathcal{L}^{\Psi}_{updt}(f) = \{H'(id), \{(\mathcal{L}^{\Sigma}_{updt}(w_i, id)) : i = 1, 2, \ldots, n_{id}\}\}$.

1. For each keyword $w_i \in f$
   (a) $\widetilde{\tau}^i_u \leftarrow \mathtt{Sim}_{\Sigma}(\mathcal{L}^{\Sigma}_{updt}(w, id))$
   (b) $\widetilde{tag}_{w_i} \xleftarrow{o} \widetilde{T}_{tag}[w_i]$
   (c) $n_{w_i} \leftarrow C[w_i] + 1$
   (d) If $T_F[(\widetilde{tag}_{w_i}, id||(n_{w_i}+1))]$ is not null,
      i. $\widetilde{pos}_i \leftarrow T_F[(\widetilde{tag}_{w_i}, id||(c_v+1))]$

      Else
      i. $\widetilde{pos}_i \leftarrow$ a random pos_i such that $\widetilde{T}_{sig}[pos_i]$ is null
      ii. $T_F[(\widetilde{tag}_{w_i}, id||(n_{w_i}+1))] \leftarrow \widetilde{pos}_i$
      iii. $\widetilde{T}'_{sig}[pos_i] \leftarrow 1$
   (e) $\widetilde{\sigma}_i \xleftarrow{\$} \mathsf{G}$
2. $\widetilde{pos} \leftarrow \{\widetilde{pos}_1, \widetilde{pos}_2, \ldots, \widetilde{pos}_{n_{id}}\}$
3. $\widetilde{\sigma} \leftarrow \{\widetilde{\sigma}_1, \widetilde{\sigma}_2, \ldots, \widetilde{\sigma}_{n_{id}}\}$
4. Return $\widetilde{\tau}^{\Psi}_u = (\widetilde{pos}, \widetilde{\sigma})$

Since, in each entry, the signature generated in T_{sig} is of the form $g^{\alpha mr}$ and corresponding entry in $\widetilde{T}_{sig}$ is of the form $g^{\alpha r'}$, where r is pseudo-random (as R is so) and r' is randomly taken, we can say that power of g in both are indistinguishable. Hence, T_{sig} and $\widetilde{T}_{sig}$ are indistinguishable.

Besides, the indistinguishability of $\widetilde{\tau}^{\Psi}_u$, $\widetilde{\tau}^{\Psi}_s$ with respect to τ^{Ψ}_s, τ^{Ψ}_u respectively follows from the pseudo-randomness of F.

□

6.3.3.2 Soundness

We see that the cloud server can cheat the cloud in four ways only, by returning– (1) incorrect number of identifiers in R_{w_1}, (2) some altered identifier in R_{w_1}, (3) some result $R_{w'}$ of other

keyword set w_1 instead of R_{w_1} or (4) some subset of $\hat{R}_{\hat{w}}$. However, for each case, the cheating will be detected as follows.

1. Since, the client stores the frequency of each keyword as in the state of the database, it can identify incorrect frequency.

2. If any identifier is altered, m_i^j in Step 18 of Algo. 20 does not match and consequently, signature verification will be failed.

3. Signature is bounded with keywords by s_w. During proof generation at the client-side, it is regenerated. So signature verification will be failed if result set is changed.

4. Finally, if some subset of $\hat{R}_{\hat{w}}$ is returned, there will be some identifier $id \in \hat{R}_{\hat{w}}$ that is skipped in the returned set. So, the cloud server has to find at least one $w \in \hat{w}$, such that (id, w) pair does not exist. However, since this is not true, the cloud server cannot give non-membership proof for any such pair.

6.4 Performance evaluation

Here we discuss a few previous schemes and compare them with our proposed one. Since we have shown that it is trivial to get a conjunctive scheme with client storage, we are considering the schemes that have no extra client storage for verifiability. We have summarized the comparison in Table 6.2.

We see that most of the works for verifiability are based on accumulators. The static scheme [96] used two types of accumulators: One for each keyword and another for the total keyword file pair. When the size of the member set increases, generating non-membership proof takes enormous time. Thus it becomes impractical for a large database. Moreover, it does not support dynamic data. Moreover, to verify, the client needs to compute $|R_{w_1}| \times |\hat{R}_{\hat{w}}|$ number of power of g and needs *two round* of communications.

The static scheme [67] used bilinear map for verifiability. Due to the existence of an auditor, the client does not need to compute anything for verification. However, during search token generation requires $O(|\mathcal{W}|)$ amount of storage as well as communication which is large when the keyword set os large.

Mio et al. [69] is a static verifiable scheme that uses an interactive challenge-response method for verification. However, it requires $O(|\mathcal{W}|.|\mathcal{DB}|)$ cloud storage which is very large. Moreover,

in verification, it requires 2 rounds of communication. It does not discuss the case when a subset of the result is returned.

The static scheme [6] used the cuckoo hashing [75]. It similarly keeps an accumulator for each keyword and uses polynomial interpolation to prove the set intersection.

Table 6.2: Comparison with existing conjunctive search SE schemes

Scheme name	Is Dyn?	Forward Secrecy	Verifiable	client Comm Cost to verify	client Comp cost to verify	client Comm Cost to update	client Comp cost to update
[96]	×	–	✓	$O(\|\mathcal{W}\|.\|\mathcal{DB}\|) + 2\mathtt{R}$	$O(\|R_{w_1}\|.\|\hat{R}_{\hat{w}}\|)\mathtt{Ex}$	–	–
[67]	×	–	✓	$O(\|\mathcal{W}\|) + 1\mathtt{R}$	$O(\|R_{\hat{w}}\|)(\mathtt{Ex} + \mathtt{Hs})$	–	–
[69]	×	–	✓	$O(\|R_{\hat{w}}\|) + 2\mathtt{R}$	$O(\|R_{\hat{w}}\|)(\mathtt{Ex} + \mathtt{Bm})$	–	–
[6]	×	–	✓	$O(\|\hat{w}\| \log \|\mathcal{W}\|) + 1\mathtt{R}$	$O(\|\hat{w}\| \log \|\mathcal{W}\|)(\mathtt{Ex} + \mathtt{Bm} + \mathtt{Hs})$	–	–
[99]	✓	✓	×	–	–	$O(\|f\|) + 1\mathtt{R}$	$O(\|f\|)\mathtt{Hs}$
[57]	✓	×	✓	$O(2\|\hat{w}\|(\log N + 1)) + 2\mathtt{R}$	$O(3\|\hat{w}\| \log N)\mathtt{M} + O(3\|\hat{w}\|)\mathtt{Ex}$	$O(\|f\|) + 1\mathtt{R}$	$O(\|f\|)\mathtt{Ex}$
[45]	✓	×	✓	$O(\|\hat{w}\| \log N) + 1\mathtt{R}$	$O(\|\hat{w}\| \log N)(\mathtt{Ex} + \mathtt{Bm})$	$O(\|f\| \log N) + 2\mathtt{R}$	$O(\|f\| \log N)\mathtt{Ex}$
Our Scheme	✓	✓	✓	$O(\|R_{w_1}\|) + 1\mathtt{R}$	$O(\|\hat{w}\|.\|R_{w_1}\|)\mathtt{Hs}$	$O(\|f\|) + 1\mathtt{R}$	$O(\|f\|)\mathtt{Hs} + O(\|f\|)\mathtt{Ex}$

M– number of multiplications in G_T, Ex– number of exponentiations in G, R– number of rounds of communication, Hs– number of hashes. Bm– number of pairing operations. * in the complexities we considered most expensive operations only.

The dynamic scheme Li et al. [57] used an accumulator for each keyword. The accumulators are stored in the cloud in a Merkle tree that ensures the integrity of them. Updates of accumulators are not discussed. One big difference of [57] with us is that [57] computes membership proofs on the go when it requires. So, if the number of searches is high, this scheme will become slow. As in most of the verifiable dynamic schemes, it also has two rounds of communication during an update. Though [99] is forward private, it is not verifiable. It is also difficult to extend it to be verifiable.

[45] is a good dynamic scheme with verifiable support. It keeps an accumulator for each keyword and makes an accumulator tree for them. To verify whether the returned intersection set is correct it uses polynomial interpolation with FFT. This makes the computational cost higher ($O(N \log^2 N)$) for the cloud server and makes the scheme unsuitable when the number of searches is high. Moreover, the scheme is not forward private too.

In our scheme, the verification can be done via any auditor, so if the client wants, it can outsource it. So, in the table bilinear map computation is ignored. Moreover, we see that most of the schemes use no extra client storage for verifiability, but they have different cloud storage requirements.

The works, including [45], [6], etc., that use intersection method, have higher complexity as they have to find all results first. The proof includes related information which increases communication cost too.

Conclusion

In this chapter, we have proposed a conjunctive DSE scheme that is verifiable too. The design is based on another designed authentication tree DIA tree which is an efficient one. Moreover, till now in the last three chapters, we have considered data as a collection of text documents.

In the next chapters, we consider graph data. We study different queries query over encrypted outsourced graphs.

Chapter 7

The Secure Link Prediction Problem

Social networks have become an integral part of our lives. These networks can be represented as graphs with nodes being entities (members) of the network and edges representing the association between entities (members). As the size of these graphs increases, it becomes quite difficult for small enterprises and business units to store the graphs in-house. So, there is a desire to store such information in cloud servers.

In order to protect the privacy of individuals (as is now mandatory in EU and other places), data is often anonymized before storing in remote cloud servers. However, as pointed out by Backstrom *et al.* [7], anonymization does not imply privacy. By carefully studying the associations between members, a lot of information can be gleaned.

The data owner, therefore, has to store the data in encrypted form. Trivially, the data owner can upload all data in encrypted form to the cloud. Whenever some query is made, data owner has to download all data, do necessary computations and re-upload the re-encrypted data. This is very inefficient and does not serve the purpose of cloud service. Thus, we need to keep the data stored in the cloud in encrypted form in such a way that we can compute efficiently on the encrypted data.

Some basic queries for a graph are neighbor query (given a vertex return the set of vertices adjacent to it), vertex degree query (given a vertex, return the number of adjacent vertices), adjacency query (given two vertices return if there is an edge between them) etc. It is important that when an encrypted graph supports some other queries, like shortest distance queries, it should not stop supporting these basic queries.

Nowell and Kleinberg [58] first defined the link prediction problem for social networks. The link prediction problem states that given a snapshot of a graph whether we can predict which new interactions between members are most likely to occur in the near future. For example, given a node A at an instant, the link prediction problem tries to find the most likely node B with which A would like to connect at a later instant. Different types of distance metrics are used to measure the likelihood of the formation of new links. The distances are called *scores* ([58]). Nowell and Kleinberg, in [58], considered several metrics including common neighbors, Jaccard's coefficient, Adamic/Adar, preferential attachment, Katz$_\beta$ etc. For example, if A and B (with no edge between them) have a large number of common neighbors they are more likely to be connected in future. In

this chapter, for simplicity, we have considered common neighbors metric to predict the emergence of a link.

Though there is a large body of literature on link prediction, to the best of our knowledge the secure version of the problem has not been studied to date. *Secure Link Prediction (SLP)* problem computes link prediction algorithms over secure i.e., encrypted data.

Our Contribution We introduce the notion of secure link prediction and present three constructions. In particular, we ask and answer the question, "Given a snapshot of a graph $G \equiv (V, E)$ (V is the set of vertices and $E \subseteq V \times V$) at a given instant and a vertex $v \in V$, which is the most likely vertex u (such that $uv \notin E$), such that, u is a neighbor of v at a later instant and $vu \notin E$". The score-metric we consider is the number of common neighbors of the two vertices v and u. This can be used to answer the question, "Given a snapshot of a graph $G = (V, E)$ at a given instant and a vertex $v \in V$, *which are the k-most likely neighbors of v* at a later instant such that none of these k vertices were neighbors of v in G."

Note that the data owner outsources an encrypted copy of the graph G to the cloud and sends an encrypted vertex v as a query. The cloud runs the secure link prediction algorithm and returns an encrypted result, from which the client can obtain the most likely neighbor of v. The cloud knows neither the graph G nor the queried vertex v.

It is to be noted that the client has much less computational and storage capacity. We propose three schemes, (`SLP-I`, `SLP-II` and `SLP-III`), in all of which, the client takes the help of a proxy server which makes it efficient to obtain query results. At a high level:

1. `SLP-I`: is the most efficient with almost no computation at client-side and leaks only the scores to the proxy server.

2. `SLP-II`: has a little more communication at client-side compared to `SLP-I` but leaks the scores of a subset of vertices to the proxy server.

3. `SLP-III`: is a very efficient scheme with almost no computation and communication at the client-side and leaks almost nothing to the proxy. This is achieved with an extra computational and communication cost between the cloud and the proxy.

In all three schemes, the client does not leak anything, to the cloud, except the number of vertices in the graph.

We have designed the scheme in such a way that it supports link prediction query as well as basic queries. Each of the previous schemes on encrypted graph are designed to support a specific

query (for example, shortest distance query, focused subgraph query etc.). However, we have designed more general schemes that support not only link prediction query but also basic queries including neighbor query, vertex degree query, adjacency query etc.

All our schemes have been shown to be adaptively secure in real-ideal paradigm.

Further, we have analyzed the performance of the schemes in terms of storage requirement, computation cost and communication cost, and counted the execution time of the schemes assuming benchmark implementations of some underlying cryptographic primitives. we have implemented prototypes for the schemes `SLP-I` and `SLP-II`, and measured the performance with different real-life datasets to study the feasibility.

From the experiment, we see that they take 12.15s and 13.75s to encrypt whereas 8.87s and 8.59s process query for a graph with 10^2 vertices.

Organization The rest of the chapter is organized as follows. Preliminaries and cryptographic tools are discussed in Section 7.1. Section 7.2 describes our proposed scheme for `SLP-I`. Two improvements of `SLP-I`, `SLP-II` and `SLP-III`, are discussed in Section 7.3 and Section 7.4 respectively. In Section 7.5, a comparative study of the complexities of our proposed schemes is given. In Section 7.6, details of our implementation and experimental results are shown. A variant of link prediction problem $\mathtt{SLP}_k$ is introduced in Section 7.7.

7.1 Preliminaries

Let $G = (V, E)$ be a graph and $A = (a_{ij})_{N\times N}$ be its adjacency matrix where N is the number of vertices and $A[i][j] = a_{ij}$. Let λ be the security parameter. Set of positive integers $\{1, 2, \cdots, n\}$ is denoted by $[n]$. By $x \xleftarrow{\$} X$, we mean to choose a random element from the set X. $D\log$ denotes the discrete logarithm. $id : \{0,1\}^* \to \{0,1\}^{\log N}$ gives the identifiers corresponding to the vertices. Let $negl(n)$ be a negligible function (see Section 2.3) over n.

We consider two PRPs (see Section 2.6) , $F_{k_{perm}}$ and π_s, where k_{perm} and s are their keys (or seeds) respectively.

7.1.1 The Link Prediction Problem

Given $G = (V, E)$, let N_v denotes the set of vertices incident on $v \in V$. Let $score(v, u)$ be a measure of how likely the vertex v is connected to another vertex u in the near future, where

$vu \notin E$. A variant of the *Link Prediction* problem states that given $v \in V$, it returns a vertex $u \in V$ ($vu \notin E$) such that $score(v, u)$ is the maximum in $\{score(v, u) : u \in V \setminus (N_v \cup \{v\})\}$ i.e.,

$$score(v, u) \geq score(v, u'), \forall u' \in V \setminus (N_v \cup \{v\}) \tag{7.1}$$

Thus, given a vertex v, we find most likely vertex to connect with. There are various metrics to measure score like the number of common neighbors, Jaccard's coefficient, Adamic/Adar metric etc.

In this chapter, we consider $score(v, u)$ as the number of common nodes between v and u i.e., $score(v, u) = |N_v \cap N_u|$. Let A be the adjacency matrix of the graph G. If i_v and i_u are the rows corresponding to the vertices v and u respectively then, the score is the inner product of the rows i.e., $score(v, u) = \sum_{k=1}^{N} A[i_v][k].A[i_u][k]$. In this chapter we have used BGN encryption scheme to securely compute this inner product.

7.1.2 System overview

Here, we describe the system model considered for the link prediction problem and goals which we want to achieve.

System Model: In our model (see Fig. 7-1), there is a client, a cloud server, and a proxy server. Each of them communicates with others to execute the protocol.

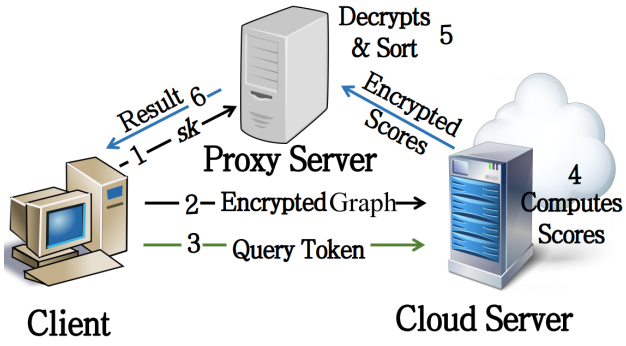


Figure 7-1: The system model of a secure link prediction scheme

- The *client* is the data owner and is considered to be *trusted*. It outsources the graph in encrypted form to the cloud server and generates link prediction queries. Given a vertex v, it queries for the vertex u which is most likely to be connected in the future.

- The *cloud server (CS)* holds the encrypted graph and computes over the encrypted data when the client requests a query. We assume that the cloud server is honest-but-curious . It is curious to learn and analyze the encrypted data and queries. Nevertheless, it is honest and follows the protocol.
- The *proxy server (PS)* helps the cloud server and the client to find the most likely vertex securely. It reduces computational overhead of the client by performing decryptions. However, the proxy server is assumed to be honest-but-curious.

All channels connecting the client, the cloud and the proxy servers are assumed to be secure. An adversary can eavesdrop on channels but cannot tamper messages sent along it. However, we assume, the cloud and the proxy servers do not collude.

This system model is to outsource as much computation as possible without leaking the information about the data, assuming the client has very low computation power (like mobile devices). This kind of model to outsource computation previously used by Wang et al. [97] for secure comparison. Assumption of the proxy and cloud server do not collude is a standard assumption.

Design Goals: In this chapter, under the assumption of the above system model, we aim at providing a solution for the secure link prediction problem. In our design, we want to achieve confidentiality, scalability, efficiency and update support over the encrypted graph, as described in Section 1.1.2.

Moreover, the client should efficiently perform neighbor query, vertex degree query or adjacency query. These are the basic query that every graph should support. The client should leak as little information as possible.

7.1.3 Secure Link Prediction Scheme

Secure Link Prediction (SLP) problem computes link prediction algorithms over secure i.e., encrypted data. In this section, we present definition of link prediction scheme for a graph G and its security against adaptive chosen-query attack.

Definition 7.1 (Secure link prediction scheme)**.** *A* secure link prediction (SLP) scheme *for a graph G is a tuple* (KeyGen, EncMatrix, TrapdoorGen, LPQuery, FindMaxVertex) *of algorithms as follows.*

- $(\mathcal{PK}, \mathcal{SK}) \leftarrow \texttt{KeyGen}(1^\lambda)$: *is a client-side PPT algorithm that takes λ as a security parameter and outputs a public key $\mathcal{PK}$ and a secret key $\mathcal{SK}$.*

- $T \leftarrow \texttt{EncMatrix}(G, \mathcal{SK}, \mathcal{PK})$: *is a client-side PPT algorithm that takes a public key $\mathcal{PK}$, a secret key $\mathcal{SK}$ and a graph G as inputs and outputs a structure T that stores the encrypted adjacency matrix of G.*
- $\tau_v \leftarrow \texttt{TrapdoorGen}(v, \mathcal{SK})$: *is a client-side PPT algorithm that takes a secret key $\mathcal{SK}$ and a vertex v as inputs and outputs a query trapdoor τ_v.*
- $\hat{c} \leftarrow \texttt{LPQuery}(\tau_v, T)$: *is a PPT algorithm run by a cloud server that takes a query trapdoor τ_v and the structure T as inputs and outputs list of encrypted scores $\hat{c}$ with all vertices.*
- $i_{res} \leftarrow \texttt{FindMaxVertex}(pk, sk, \hat{c})$: *is a PPT algorithm run by a proxy server that takes pk, sk and $\hat{c}$ as inputs and outputs the most probable vertex identifier i_{res} to connect with the queried vertex.*

Correctness: An SLP scheme is said to be correct if, $\forall \lambda \in \mathsf{N}$, $\forall(\mathcal{PK}, \mathcal{SK})$ generated using $\texttt{KeyGen}(1^\lambda)$ and all sequences of queries on T, each query outputs a correct vertex identifier except with negligible probability.

Adaptive security: An SLP scheme should have two properties:

1. Given T, the cloud servers should not learn any information about G and
2. From a sequence of query trapdoors, the servers should learn nothing about corresponding queried vertices.

The security of an SLP is defined in real-ideal paradigm. In real scenario, the challenger $\mathcal{C}$ generates keys. The adversary $\mathcal{A}$ generates a graph G which it sends to $\mathcal{C}$. $\mathcal{C}$ encrypts the graph with its secret key and sends it to $\mathcal{A}$. Later, q times it finds a query vertex based on previous results (i.e., adaptive) and receives trapdoor for the current. Finally $\mathcal{A}$ outputs a guess bit b. In ideal scenario, on receiving the graph G, the simulator $\mathcal{S}$ generates a simulated encrypted matrix. For each adaptive query of $\mathcal{A}$, $\mathcal{S}$ returns a simulated token. Finally $\mathcal{A}$ outputs a guess bit b'. The security definition (Definition 7.2) ensures $\mathcal{A}$ cannot distinguish $\mathcal{C}$ from $\mathcal{S}$.

We have assumed that the communication channel between the client and the servers are secure. Since the CS and the PS do not collude, they do not share their collected information. So, the simulator can treat CS and PS separately.

In our scheme, the proxy server does not have the encrypted data or the trapdoors. During query operation, it gets a set of scrambled scores of the queried vertex with other vertices. So, we can consider only the cloud server as the adversary (see [14]). Let us define security as follows.

Algorithm 24: $\mathbf{Real}_{\mathcal{A}}^{\mathtt{SLP}}(\lambda)$

$(\mathcal{PK}, \mathcal{SK}) \leftarrow \mathtt{KeyGen}(1^\lambda)$
$(G, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$T \leftarrow \mathtt{EncMatrix}(G, \mathcal{SK}, \mathcal{PK})$
$(v_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, T)$
$\tau_{v_1} \leftarrow \mathtt{TrapdoorGen}(v_1, \mathcal{SK})$
for $2 \leq i \leq q$ **do**
  $(v_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, T, \tau_{v_1}, \ldots, \tau_{v_{i-1}})$
  $\tau_{v_i} \leftarrow \mathtt{TrapdoorGen}(v_i, \mathcal{SK})$
end
$\tau = (\tau_{v_1}, \tau_{v_2}, \ldots, \tau_{v_q})$
$b \leftarrow \mathcal{A}_{q+1}(T, \tau, st_{\mathcal{A}})$, where $b \in \{0, 1\}$
return b

Algorithm 25: $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathtt{SLP}}(\lambda)$

$(G, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$(st_{\mathcal{S}}, T) \leftarrow \mathcal{S}_0(\mathcal{L}_{bld}(G))$
$(v_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, T)$
$(\tau_{v_1}, st_{\mathcal{S}}) \leftarrow \mathcal{S}_1(st_{\mathcal{S}}, \mathcal{L}_{qry}(v_1))$
for $2 \leq i \leq q$ **do**
  $(v_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, T, \tau_{v_1}, \ldots, \tau_{v_{i-1}})$
  $(\tau_{v_i}, st_{\mathcal{S}}) \leftarrow \mathcal{S}_i(st_{\mathcal{S}}, \mathcal{L}_{qry}(v_1), \ldots, \mathcal{L}_{qry}(v_{i-1}))$
end
$\tau = (\tau_{v_1}, \tau_{v_2}, \ldots, \tau_{v_q})$
$b' \leftarrow \mathcal{A}_{q+1}(T, \tau, st_{\mathcal{A}})$, where $b' \in \{0, 1\}$
return b'

Definition 7.2 (Adaptive semantic security (CQA2) of a link prediction scheme)**.** *Let* SLP = (KeyGen, EncMatrix, TrapdoorGen, LPQuery, FindMaxVertex) *be a secure link prediction scheme. Let $\mathcal{A}$ be a stateful adversary, $\mathcal{C}$ be a challenger, $\mathcal{S}$ be a stateful simulator and $\mathcal{L} = (\mathcal{L}_{bld}, \mathcal{L}_{qry})$ be a stateful leakage algorithm. Let us consider two games- $\mathbf{Real}_{\mathcal{A}}^{\mathtt{SLP}}(\lambda)$ (see Algo. 24) and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathtt{SLP}}(\lambda)$ (see Algo. 25).*

The SLP *is said to be adaptively semantically $\mathcal{L}$-secure against chosen-query attacks (*CQA2*) if, $\forall$ PPT adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_{q+1})$, where $q = poly(\lambda)$, $\exists$ a PPT simulator $\mathcal{S} = (\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_q)$, such that*

$$|Pr[\mathbf{Real}_{\mathcal{A}}^{\mathtt{SLP}}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathtt{SLP}}(\lambda) = 1]| \leq negl(\lambda) \tag{7.2}$$

7.1.4 Overview of our proposed schemes

A graph can be encrypted in several ways like adjacency matrix, adjacency list, edge list etc. Each of them has advantages and disadvantages depending on the application. In our scheme, we have defined score as the number of common neighbors that can be calculated just by computing inner product of the rows corresponding to the calculating vertices. The basic idea is that, given a vertex, to predict the most probable vertex to connect with, we compute scores with all other vertices and sort them according to their score. However, calculating the inner product and sorting them in cloud server are expensive operations and there is no scheme that provides all of the functionality

to be computed over encrypted data. So, we have used BGN homomorphic encryption scheme that enables us to compute inner product on encrypted data. Choosing BGN, gives power to the client for querying not only link prediction query but also neighbor query, degree of a vertex query, adjacency query etc.

Besides, the score computation, the score decryption and sorting the score in encrypted form is non-trivial keeping in mind that the client has low computation power. So, we have proposed three schemes that perform score computations as well as sorting on encrypted data with the help of a honest-but-querious proxy server which does not collude with the cloud server. The three schemes show tread-off between the computation cost, communication cost and leakage in order to compute the vertex most probable to connect with.

7.2 Our proposed protocol for SLP

In this section, we propose an efficient scheme SLP-I and analyze its security. The scheme is divided into three phases– key generation, data encryption, and query phase. The client first generates required secret and public keys. Then it encrypts the adjacency matrix of the graph in a structure and uploads it to the CS. To query for a vertex, the client generates a query trapdoor and sends it to the CS. The CS computes encrypted score (i.e., inner products of the row corresponding to the queried vertex with the other vertices on the encrypted graph). The PS decrypts the scores, finds the vertex with highest score and sends the result to the client.

Key Generation: In this phase, given a security parameter λ, the client chooses a bilinear map $e : \mathsf{G} \times \mathsf{G} \rightarrow \mathsf{G}_1$. Then, the permutation key k_{perm} is chosen at random for the PRP $F : \{0,1\}^* \times \{0,1\}^{\log N} \rightarrow \{0,1\}^{\log N}$. It executes $\mathtt{BGN.Gen}()$ to get sk and pk. After generating private key $\mathcal{SK}$ and public key $\mathcal{PK}$, a part sk of $\mathcal{SK}$ is shared with the PS. This part of the key helps the PS to compute secure comparisons. Key generation is described in Algo. 26.

Algorithm 26: $\mathtt{KeyGen}(1^\lambda)$

$k_{perm} \xleftarrow{\$} \{0,1\}^\lambda$
$(pk, sk) \leftarrow \mathtt{BGN.Gen}(1^\lambda)$
$\mathcal{PK} \leftarrow pk$; $\mathcal{SK} \leftarrow (sk, k_{perm})$
return $(\mathcal{PK}, \mathcal{SK})$

Data Encryption: In this phase, the client encrypts the adjacency matrix with its private key and uploads the encrypted matrix to the CS (see Algo. 27). Each entry a_{ij} in the adjacency matrix A of

G is encrypted using Algo. 2. Let $M = (m_{ij})_{N\times N}$ be the encrypted matrix. Then, each row of M is stored in the structure T. The PRP F gives the position in T corresponding to vertices. Finally, the structure T is sent to the CS.

Algorithm 27: $\texttt{EncMatrixI}(A, \mathcal{SK}, \mathcal{PK})$

$(n, \mathsf{G}, \mathsf{G}_1, e, g, h) \leftarrow \mathcal{PK}$
$(q_1, k_{perm}) \leftarrow \mathcal{SK}$
for $i = 1, j = 1$ **to** $i = N, j = N$ **do**
  $m_{ij} \leftarrow \texttt{BGN.Encrypt}_{\mathsf{G}}(\mathcal{PK}.pk, a_{ij})$
end
Construct a structure T of size N.
for $i = 1$ **to** $i = N$ **do**
  $ind \leftarrow F_{k_{perm}}(id(v_i))$
  $T[ind] \leftarrow (m_{i1}, m_{i2}, \ldots, m_{iN})$.
end
return T

Algorithm 28: $\texttt{TrapdoorGenI}(v, \mathcal{SK})$

$(sk, k_{perm}) \leftarrow \mathcal{SK}$
$i' \leftarrow F_{k_{perm}}(id(v))$; $s \xleftarrow{\$} \{0,1\}^{\lambda}$
$\tau_v \leftarrow (i', s)$
return τ_v

Algorithm 29: $\texttt{LPQueryI}(\tau_v, T)$

$N \leftarrow |T|$; $(i', s) \leftarrow \tau_v$
$(m_{i'1}, m_{i'2}, \ldots, m_{i'N}) \leftarrow T[i']$
for $i = 1$ **to** $i = N$ **do**
  $r \xleftarrow{\$} \{0,1\}^{\lambda}$
  if $i \neq i'$ **then**
    $(m_{i1}, m_{i2}, \ldots, m_{iN}) \leftarrow T[i]$
    $c_i \leftarrow e(g,h)^r . \prod_{k=1}^{N} e(m_{i'k}, m_{ik})$
  else
    $c_{i'} \leftarrow e(g,g)^0 . e(g,h)^r$
  end
end
$\pi_s \leftarrow$ permutation with key s.
$\hat{c} \leftarrow (c_{\pi_s(1)}, c_{\pi_s(2)}, \ldots, c_{\pi_s(N)})$
$\hat{m} \leftarrow (m_{\pi_s(1)}, m_{\pi_s(2)}, \ldots, m_{\pi_s(N)})$,
where $m_i \leftarrow m_{i'i} . h^{r_i}$, $r_i \xleftarrow{\$} \{0,1\}^{\lambda}$
return $(\hat{c}, \hat{m})$ to the PS

Query: In the query phase, the client sends a query trapdoor to the CS. The CS finds encrypted scores with respect to the other vertices and sends them to the PS. The PS decrypts them and sends the identifier of the vertex with highest score to the client.

To query for a vertex v, the client first chooses a secret key $s \xleftarrow{\$} \{0,1\}^{\lambda}$ for the PRP π_s that is not known to the PS (see Algo. 28). Then it finds the position $i' = F_{k_{perm}}(id(v))$. Finally, the client sends the trapdoor $\tau_v = (i', s)$ as query trapdoor to the CS.

On receiving τ_v, the CS computes the encrypted scores $(c_1, c_2, \ldots, c_N)$ (see Algo. 29) and computes $(m_1, m_2, \ldots, m_N)$ corresponding to the queried vertex. Using π_s, the CS shuffles the order of the encrypted scores and m_i's. Finally, the CS sends the shuffled encrypted scores and the scrambled queried-row entries $(m_{\pi_s(1)}, m_{\pi_s(2)}, \ldots, m_{\pi_s(N)})$ to the PS.

Since, the PS has sk $(= q_1)$, it can decrypt all $\bar{c}_i$s and $\bar{m}_i$s. It decrypts $\bar{m}_i$ first and then decrypts $\bar{c}_i$ only if corresponding decrypted value of $\bar{m}_i$ is 0. Then, it takes an i_{res} such that $s_{i_{res}}$ is the maximum in the set $\{s_i : i \in [N]\}$ and sends it to the client (see Algo. 30). Finally, the client finds the resulting vertex identifier v_{res} as $v_{res} \leftarrow \pi_s^{-1}(i_{res})$.

Algorithm 30: `FindMaxVertexI`$(sk, \hat{c}, \hat{m})$

$(\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_N) \leftarrow \hat{c}$
$(\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_N) \leftarrow \hat{m}$
for $i = 1$ **to** $i = N$ **do**
  $s_i \leftarrow \texttt{BGN.Decrypt}_{\texttt{G}_1}(pk, sk, \bar{c}_i)$
  $a_i \leftarrow (\texttt{BGN.Decrypt}_{\texttt{G}}(pk, sk, \bar{m}_i)) \bmod 2$
end
$i_{res} \leftarrow i : (a_i = 0) \wedge (s_i = max\{s_j : j \in [N]\})$
return i_{res} to the client

Correctness: For any two rows $T[i]$ and $T[j]$, if c_{ij} is the encryption of the score s_{ij} then, $c_{ij} = e(g,h)^r \prod_{k=1}^{N} e(m_{ik}, m_{jk})$. Again, since $e(g,g)^{q_1 q_2} = 1$, we get $(c_{ij})^{q_1} = (e(g,g)^{q_1})^{\sum_{k=1}^{N} a_{ik} a_{jk}} = \hat{g}^{s_{ij}}$, *where* $\hat{g} = e(g,g)^{q_1}$.

Thus, $D\log$ of $(c_{ij})^{q_1}$ to the base $\hat{g}$ gives s_{ij}. If powers of $\hat{g}$ are pre-computed, the score s_{ij} can be found in constant time. However, Pollard's lambda method [64] can be used to find discrete logarithm of c'_{ij} base $\hat{g}$.

Security analysis

In the security definition, a small amount of leakage has been allowed. The adversary knows the algorithms and possesses the encrypted data and queried trapdoors. Only $\mathcal{SK}$ is unknown to it. The leakage function $\mathcal{L}$ is a pair $(\mathcal{L}_{bld}, \mathcal{L}_{qry})$ (associated with `EncMatrix` and `LPQuery` respectively) where $\mathcal{L}_{bld}(G) = \{|T|\}$ and $\mathcal{L}_{qry}(v) = \{\tau_v\}$.

Theorem 7.1. *If* `BGN` *is semantically secure and* F *is a PRP, then* `SLP-I` *is* $\mathcal{L}$*-secure against adaptive chosen-query attacks.*

Proof. The proof of security is based on the simulation-based `CQA-II` security (see Definition 7.2). Given the leakage $\mathcal{L}_{bld}$, the simulator $\mathcal{S}$ generates a randomized structure $\widetilde{T}$ which simulates the structure T of the challenger $\mathcal{C}$. Given a query trapdoor τ_v, $\mathcal{S}$ returns simulated trapdoors $\widetilde{\tau_v}$ maintaining system consistency of the future queries by the adversary. To prove the theorem, it is enough to show that the trapdoors generated by $\mathcal{C}$ and $\mathcal{S}$ are indistinguishable to $\mathcal{A}$.

- (Simulating the structure T) $\mathcal{S}$ first generates $(\mathcal{SK}, \mathcal{PK}) \leftarrow \texttt{BGN.Gen}(1^\lambda)$. Given $\mathcal{L}_{bld}(A)$, $\mathcal{S}$ takes an empty structure $\widetilde{T}$ of size $|T|$.
  Finally, it takes $\widetilde{m_{ij}} \leftarrow \texttt{BGN.Encrypt}_{\{\texttt{mathbbG}\}}(\mathcal{PK}.pk, 0^\lambda)$, $(i,j) \in [N] \times [N]$ where $N = |T|$.

- (Simulating query trapdoor τ_v) $\mathcal{S}$ first takes an empty dictionary Q. Given $\mathcal{L}_{srch}(v)$, $\mathcal{S}$ checks whether v is present in Q. If not, it takes a random $\log N$-bit string $\widetilde{\tau_v}$, stores it as $Q[v] = \widetilde{\tau_v}$ and returns $\widetilde{\tau_v}$. If v has appeared before, it returns $Q[v]$.

Semantic security of BGN guarantees that $\widetilde{m_{ij}}$ and m_{ij} are indistinguishable. Since F is a PRP, $\widetilde{\tau_v}$ and τ_v are indistinguishable. This completes the proof. □

7.3 SLP-II with less leakage

Though the SLP-I scheme is efficient, it has few disadvantages. Firstly, in SLP-I, the number of common nodes between the queried vertex and all other vertices are leaked to the PS which provides partial knowledge of the graph to it. Since, the server PS is semi honest, we want to leak as little information as possible. In this section, we propose another scheme SLP-II that hides most of the scores from the PS which results in leakage reduction.

Secondly, the client has high communication cost with PS while processing a link prediction query. Our proposed SLP-II scheme has an advantages over this with reduced communication cost from CS to PS is. We achieve these by using extra storage of size of the matrix M and extra bandwidth from the PS to the CS of $O(N)$.

7.3.1 Proposed protocol

In SLP-II, after computing the scores, the CS increases that of the incident vertices randomly from maximum possible score i.e., degree of the queried vertex. For example, let s be a score in the form g_1^s, then a random number r, greater than or equal to the degree, is added with it. Then the scores is increased as $g_1^s.g_1^r = g_1^{(s+r)}$. Since lower bound of r is known to the client, it can eliminate the scores with adjacent vertices. The PS only derypts the scores and sends the sorted list to the client. Since the degree is hidden from PS and known to the client, it can eliminate the vertices with score larger than degree. The algorithms are as follows.

Key Generation: Same as Algo. 26.

Data Encryption: In SLP-II, data encryption is similar to Algo. 27. Together with $M = (m_{ij})_{N\times N}$, another matrix $M' = (m'_{ij})_{N\times N}$ is generated by encrypting a matrix B (see Algo. 31). The matrix $B = (b_{ij})_{N\times N}$ where $b_{ij} = t$, $(\deg v_i < t < N - \deg v_i)$ if v_i and v_j are connected, else $b_{ij} = 0$. Now, $m'_{ij} = e(g,g)^{b_{ij}}.e(g,h)^{r_{ij}}$, where notations are usual. Finally, The matrices M and M' are

uploaded to the CS together in structures T and T' respectively. Rows of M and M' corresponding to the vertex v are stored in $T[F_{k_{perm}}(id(v))]$ and $T'[F_{k_{perm}}(id(v))]$ respectively. Note that, entries of M are in the group G whereas that of M' are in G_1.

Algorithm 31: $\texttt{EncMatrixII}(A, \mathcal{SK}, \mathcal{PK})$

1 $(n, \mathsf{G}, \mathsf{G}_1, e, g, h) \leftarrow \mathcal{PK}$;
$(q_1, k_{perm}) \leftarrow \mathcal{SK}$
2 Construct matrix B from A
3 **for** $i = 1, j = 1$ **to** $i = N, j = N$ **do**
4 $\quad m_{ij} \leftarrow \texttt{BGN.Encrypt}_{\mathsf{G}}(\mathcal{PK}.pk, a_{ij})$
5 $\quad m'_{ij} \leftarrow \texttt{BGN.Encrypt}_{\mathsf{G}_1}(\mathcal{PK}.pk, b_{ij})$
6 **end**
7 Construct structures T and T' of size N
8 **for** $i = 1$ **to** $i = N$ **do**
9 $\quad ind_i \leftarrow F_{k_{perm}}(id(v_i))$
10 $\quad T[ind_i] \leftarrow (m_{i1}, m_{i2}, \ldots, m_{iN})$
11 $\quad T'[ind_i] \leftarrow (m'_{i1}, m'_{i2}, \ldots, m'_{iN})$
12 **end**
13 **return** (T, T')

Algorithm 32: $\texttt{LPQueryII}(\tau_v, T)$

1 $N \leftarrow |T|$; $(i', s) \leftarrow \tau_v$
2 $(m_{i'1}, m_{i'2}, \ldots, m_{i'N}) \leftarrow T[i']$
3 **for** $i = 1$ **to** $i = N$ **do**
4 $\quad r \xleftarrow{\$} \{0,1\}^{\lambda}$
5 $\quad$ **if** $i \neq i'$ **then**
6 $\quad\quad (m_{i1}, m_{i2}, \ldots, m_{iN}) \leftarrow T[i]$
7 $\quad\quad c_i \leftarrow e(g,h)^r . \prod_{k=1}^{N} e(m_{i'k}, m_{ik})$
8 $\quad$ **else**
9 $\quad\quad c_i \leftarrow e(g,g)^0 . e(g,h)^r$
10 $\quad$ **end**
11 $\quad c_i = c_i . m'_{i'i}$
12 **end**
13 $m \leftarrow \prod_{i=1}^{i=N} m_{i'i}$
14 $\pi_s \leftarrow$ permutation with key s.
15 $\hat{c} \leftarrow (c_{\pi_s(1)}, c_{\pi_s(2)}, \ldots, c_{\pi_s(N)})$
16 **return** $\hat{d}$ to PS and m to the client

Query: As in the previous scheme, the client sends query trapdoor $\tau_v = (i', s)$ to the CS for a vertex v. Let $\hat{c} = (c_1, c_2, \ldots, c_N)$ be the set of encrypted scores computed in step 7 of Algo. 32. In addition, for each i, c_i is updated as $c_i = c_i . m'_{i'i}$. Then $\hat{c} = (c_{\pi_s(1)}, c_{\pi_s(2)}, \ldots, c_{\pi_s(N)})$ is sent to the PS. Instead of sending $\hat{m}$ to the PS, $m = \prod_{i=1}^{i=N} m_{i'i}$ is sent to the client, which results the encryption of the degree of the vertex v. `SLP-II` query is described in Algo. 32.

The PS decrypts $\hat{c}$ as $s'_1, s'_2, \ldots, s'_N$ and sorts them. Then, the PS sends (s'_{i_1}, i_1), (s'_{i_2}, i_2), …, (s'_{i_N}, i_N) where s'_{i_j}'s are in sorted order and i_j's are their indices in $\hat{c}$ (see Algo.33).

The client takes the first index $i_{res} = i_j$ such that $s'_{i_j} \leq \deg v$. The client gets $\deg v$ by decrypting m. Finally, the client can find the resulting vertex identifier v_{res} as $v_{res} \leftarrow \pi_s^{-1}(i_{res})$.

Correctness: For all i, the decrypted entry s'_i (line 3, Algo. 33) is equals to $s_i + b_{i'i}$ where s_i is the actual score. Since $s_i \leq \deg v$ and $b_{i'i}$ is zero, when $v_{i'}$ and v_i are connected, we can see that, s'_i becomes greater than $\deg v$ when $v_{i'}$ and v_i are connected. So, the client can eliminate these entries from the list.

Algorithm 33: FindMaxVertexII$(sk, \bar{c}, \bar{m})$

$(\bar{d}_1, \bar{d}_2, \ldots, \bar{d}_N) \leftarrow \hat{d}$
for $i = 1$ **to** $i = N$ **do**
  $s'_i \leftarrow \mathtt{BGN.Decrypt}_{\mathtt{G}_1}(pk, sk, \bar{d}_i)$
end
Sorting s'_is gets $((s'_{i_1}, i_1), (s'_{i_2}, i_2), \ldots, (s'_{i_N}, i_N))$
return $((s'_{i_1}, i_1), (s'_{i_2}, i_2), \ldots, (s'_{i_N}, i_N))$

Security analysis

SLP-II does not leak any extra information to the CS than SLP-I. The leakage $\mathcal{L} = (\mathcal{L}_{bld}, \mathcal{L}_{qry})$ is same as it is in SLP-I.

Theorem 7.2. *If* BGN *is semantically secure and F is a PRP, then* SLP-II *is $\mathcal{L}$-secure against adaptive chosen-query attacks.*

Proof. As we have seen the proof of Theorem 7.1, The simulator requires to simulate the T, T' and τ_v. To simulate the structure T', given $\mathcal{L}_{bld}(A)$, $\mathcal{S}$ takes an empty structure $\widetilde{T}'$ of size $|T'|$. Finally, it takes $\widetilde{m'_{ij}} \leftarrow \mathtt{BGN.Encrypt}_{\mathtt{G}_1}(\mathcal{PK}.pk, 0^\lambda)$, $(i, j) \in [N] \times [N]$. Rest of the proof is similar as that of Theorem 7.1. □

7.4 SLP scheme using garbled circuit (SLP-III)

In SLP-II, the PS is still able to get scores with many vertices and there is a good amount of communication cost from PS to the client. In this section, we propose SLP-III in which PS does not get any scores. Besides, the proxy needs to send only result to the client which reduces communication overhead for the client.

7.4.1 Protocol description

In SLP-III, after generating the keys, the client encrypts the adjacency matrix of the graph and uploads it to the CS. At the same time, it shares a part of its secret key with the PS. In the query phase, the CS computes the encrypted scores on receiving query trapdoor from the client. However, it masks each score with random number selected by itself before sending them to the PS. The PS decrypts the masked scores and evaluates a garbled circuit, constructed by the CS (as de-

scribed in Section 7.4.2), to find the vertex with maximum score. Finally, the PS returns the index corresponding to the evaluated identifier of the vertex with maximum score.

Key Generation: Same as Algo. 26.

Data Encryption: Same as Algo. 27.

Query: To query for a vertex v, the client generates a query trapdoor $t_v = (i', s)$ (see Algo. 28) and sends it to the CS. On receiving τ_v, the CS computes the encrypted scores $(c_1, c_2, \ldots, c_N)$. It then considers the row $T[i'] = (m_{i'1}, m_{i'2}, \ldots, m_{i'N})$ corresponding to the queried vertex. Then, with random r_i and r'_i, it computes, $\bar{c}_i \leftarrow c_{\pi_s(i)}.\texttt{BGN.Encrypt}_{\mathsf{G}_1}(\mathcal{PK}.pk, r_i)$ and $\bar{m}_i \leftarrow m_{i'\pi_s(i)}.\texttt{BGN.Encrypt}_{\mathsf{G}}(\mathcal{PK}.pk, r'_i)$, for all i. If the encrypted scores are sent directly, the PS can decrypt the scores directly as it has the partial secret key sk. That is why the CS chooses random r_is and r'_is to mask them.

Algorithm 34: $\texttt{LPQueryIII}(\tau_v, T)$

$N \leftarrow |T|$; $(i', s) \leftarrow \tau_v$
$(m_{i'1}, m_{i'2}, \ldots, m_{i'N}) \leftarrow T[i']$
for $i = 1$ **to** $i = N$ **do**
  if $i \neq i'$ **then**
    $(m_{i1}, m_{i2}, \ldots, m_{iN}) \leftarrow T[i]$
    $c_i \leftarrow \prod_{k=1}^{N} e(m_{\tau_v k}, m_{ik})$
  else
    $r \xleftarrow{\$} \{0,1\}^\lambda$
    $c_{i'} \leftarrow e(g,g)^0.e(g,h)^r$
  end
end
$\pi_s \leftarrow$ permutation with key s.
for $i = 1$ **to** $i = N$ **do**
  $r_i, r'_i, x_i, x'_i \xleftarrow{\$} \{0,1\}^\lambda$
  $\bar{c}_i \leftarrow c_{\pi_s(i)}.e(g,g)^{r_i}.e(g,h)^{x_i}$
  $\bar{m}_i \leftarrow m_{i'\pi_s(i)}.g^{r'_i}.h^{x'_i}$
end
$\hat{c} \leftarrow (\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_N)$
$\hat{m} \leftarrow (\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_N)$
Computes MGC
return ($\hat{c}$, $\hat{m}$, MGC) to PS

To find the vertex with highest score, the CS builds a garbled circuit MGC (see Fig. 7-2) as described in Section 7.4.2. The CS sends $\hat{c} = (\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_N)$ and $\hat{m} = (\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_N)$ together with a garbled circuit MGC. The CS-side algorithm is described in Algo. 34.

The PS receives $\hat{c}$ and $\hat{m}$. $\forall i$, let $\bar{s}_i$ and $\bar{a}_i$ be the decryption of $\bar{c}_i$ and $\bar{m}_i$ respectively (see Algo. 35). Then, the PS evaluates MGC. During evaluation, the PS gives all $\bar{s}_i$s and a_is and corresponding indices is as input where $a_i = (\bar{a}_i \bmod 2)$. The CS gives r_is and r''_is where $r''_i = (r'_i \bmod 2)$, $\forall i$ (see Section 7.4.2).

From MGC, the PS gets an index i_{res} which is sent to the client. Finally, the client finds the resulting vertex identifier v_{res} as $v_{res} \leftarrow \pi_s^{-1}(i_{res})$.

Algorithm 35: `FindMaxVertexIII`$(sk, \hat{c}, \hat{m}, GC)$

$(\bar{c}_1, \bar{c}_2, \ldots, \bar{c}_N) \leftarrow \hat{c}$
$(\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_N) \leftarrow \hat{m}$
for $i = 1$ **to** $i = N$ **do**
    $\bar{s}_i \leftarrow \texttt{BGN.Decrypt}_{\mathsf{G}_1}(pk, sk, \bar{c}_i)$
    $\bar{a}_i \leftarrow (\texttt{BGN.Decrypt}_{\mathsf{G}}(pk, sk, \bar{m}_i))$
    $a_i \leftarrow \bar{a}_i \bmod 2$
end
Evaluates MGC with $\bar{s}_i$ and a_is as its inputs.
$i_{res} \leftarrow$ output of the MGC evaluation
return i_{res} to the client

7.4.2 Maximum Garbled Circuit (MGC)

We want minimum information to be leaked to both the servers. Without the knowledge of values, it is hard to find the maximum value because it is an iterative comparison process and requires several round of communication if we use only secure comparison. However, building a maximum garbled circuit allows cloud and proxy servers to find the maximum without knowing the value by anyone.

Kolesnikov and Schneider [51] first presented a garbled circuit that computes minimum from a set of distance. In their scheme, one party holds a set of points and the second party holds a single point. They used homomorphic encryption to compute the distances from the single points to the set of points and sort them using the garble circuit. However, *the original value of the points belongs to them were known to them*. In this chapter, we have introduced a novel maximum garbled circuit (MGC) by which one party computes the maximum from a set of numbers, *without the knowledge their values*, with the help of another party without leaking them to it. Given a set of scores MGC outputs only the identity of the vertex with maximum score.

Computing vertex with max score: In SLP-III, the CS computes a garbled circuit MGC (an example is shown in Fig. 7-2) for each query to find the maximum scored vertex identifier. Before computing MGC, in SLP-III, the PS gets $(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_N)$ and $(a_1, a_2, \ldots, a_N)$ (Algo. 35). The CS keeps $(r_1, r_2, \ldots, r_N)$ and $(r''_1, r''_2, \ldots, r''_N)$ which are used as input in MGC. During construction, it keeps the indices in the MGC such a way that MGC outputs only the index of the resulted maximum score.

MGC is required to find the index corresponding to the maximum scored vertex. The circuit is constructed layer by layer. The idea is to compare pair of scores every time in a layer and pass

the result for the next until the resulted vertex is found. If $|V| = N$, MGC has $(\log N + 1)$ layers starting from 0 to N. In the 0th layer, there are N number of NSS blocks and the rest of the blocks are Max block. The NSS blocks is for the 1st layers and computes the scores securely without knowing them. Thus, each NSS block corresponds to some vertex. Max computes the maximum score and corresponding index without knowing them. Example of a MGC, to compute maximum, assuming $N = 7$ and using Max blocks and NSS blocks, is shown in Fig. 7-2. MGC for any N is constructed similarly.

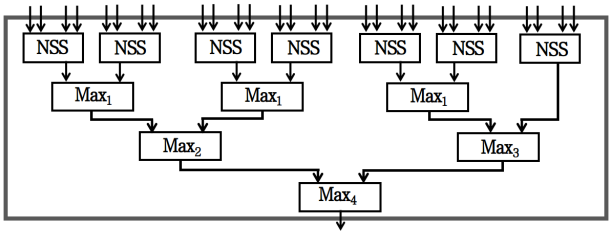
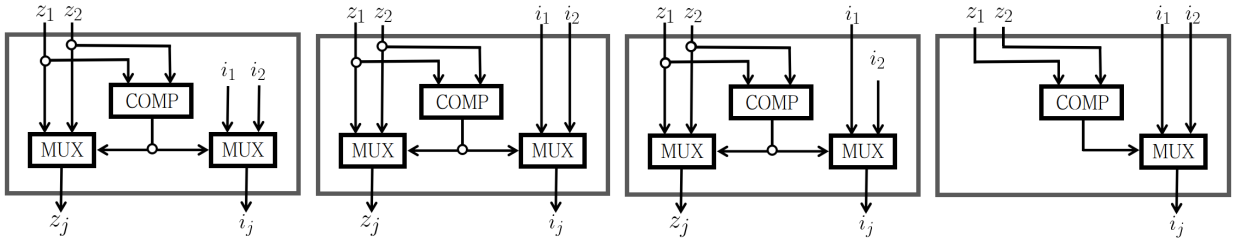


Figure 7-2: Example of a Maximum circuit with $N = 7$

Max blocks There are 4-types of Max blocks to compute the maximum- $\texttt{Max}_1$, $\texttt{Max}_2$, $\texttt{Max}_3$ and $\texttt{Max}_4$ (see Fig. 7-3). The blocks are made different to handle extreme cases. These blocks use COMP and MUX blocks (see Section 2.18).

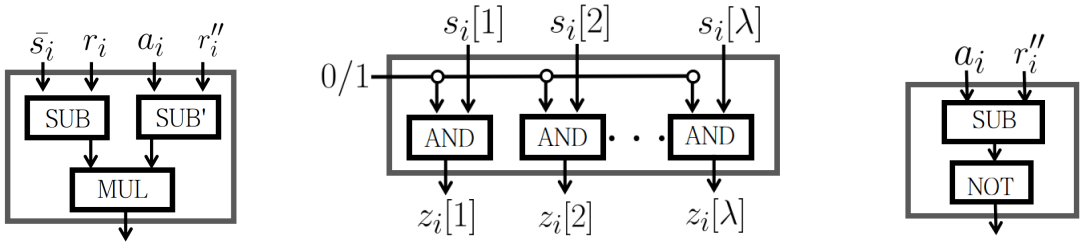


(a) $\texttt{Max}_1$ block (b) $\texttt{Max}_2$ block (c) $\texttt{Max}_3$ block (d) $\texttt{Max}_4$ block

Figure 7-3: Different max blocks used in MAXIMUM circuit

NSS blocks: Each NSS block has four inputs $\bar{s}_i$, r_i, a_i and r_i''. The inputs r_i and r_i'' comes from the CS while $\bar{s}_i$ and a_i comes from the PS. It first subtracts r_i from $\bar{s}_i$ using SUB block to get the score s_i. Then, using $\texttt{SUB}'$ block, it finds the flag bit that tells whether the vertex is adjacent to the queried vertex. MUL block (see Fig 7-4b) is used in NSS block as shown in Fig. 7-4a to make the score s_i zero if the vertex is adjacent else keeps the score s_i same.

Elimination of scores for adjacent vertices: It can be seen from encryption that $\bar{s}_i = s_i + r_i$, where s_i is the actual score corresponding to ith row and r_i randomizes the score. Each bit r_i'' is



(a) NSS block (b) MUL block (c) SUB′ block

Figure 7-4: Few circuit blocks

taken to indicate whether r'_i is odd or even. On the other hand, each bit a_i indicates whether the decrypted $\bar{a}_i$ is odd or even. Inequality of r''_i and a_i indicates that the vertex corresponding to ith row is connected with the queried vertex. In that case, we consider the score $s_i = 0$.

The block SUB′, in Fig. 7-4c, finds outputs 1 if they are equal, else outputs 0. Since, $(\bar{s}_i - r_i)$ gives the score, SUB block (see Section. 2.18) is used in MGC to compute the scores where the PS gives $\bar{s}_i$ and CS gives r_i. It can be seen that SUB′ subtract only one bit which is very efficient.

7.4.3 Security analysis

In SLP-III (as descried in Section 7.4.1), though the PS has almost no leakage, the CS has a little more leakage than SLP-I. This extra leakage occurs when it interacts with the PS through OT protocol to provide encoding corresponding to the input of PS. Since OT is secure and does not leak any meaningful information, we can ignore this leakage. In SLP-III, the leakage $\mathcal{L} = (\mathcal{L}_{bld}, \mathcal{L}_{qry})$ is same as it is in SLP-I.

Theorem 7.3. *If* BGN *is semantically secure and F is a PRP, then* SLP-III *is $\mathcal{L}$-secure against adaptive chosen-query attacks.*

Proof. The proof is the same as that of Theorem 7.1. □

7.4.4 Basic queries

All the three schemes support basic queries which includes neighbor query, vertex degree query and adjacency query.

Neighbor query: Given a vertex, neighbor query is to return the set of vertices adjacent to it. It is

to be noted that, since we have encrypted adjacency matrix of the graph, it is enough for the client if it gets the decrypted row corresponding to the queried vertex,

To query neighbor for a vertex v, the client generates $\tau_v = (i', s)$ as in Algo. 28 and sends it to the CS. The CS permutes rows corresponding to row i' and send the permuted row $\hat{m} \leftarrow (m_{\pi_s(1)}, m_{\pi_s(2)}, \ldots, m_{\pi_s(N)})$ to the PS. The PS decrypts them and send the decrypted vector $(a_1, a_2, \ldots, a_N)$ to the client. The client can compute inverse permutation for the entries for which the entries are 1. Here, the CS gets only the queried vertex and the PS gets the degree of the vertex.

Vertex degree query: To query degree of a vertex v, similarly, the client sends $\tau_v = i'$ to the CS. The CS computes encrypted degree as $m \leftarrow \prod_{i=1}^{i=N} m_{i'i}$ and sends m to the proxy. The proxy decrypts m and sends the result to the client. s is not needed as permuting the elements of some row is not required.

Here, the degree is leaked to the PS which can be prevented by randomizing the result. The CS can randomize the encrypted degree and send the randomization secret to the client. The client can get the degree just by subtracting the randomization from the result by the PS.

However, this leakage can be avoided easily, without randomizing the encrypted degree, if the client performs the decryption.

Adjacency Query: Given two vertices, adjacency query (edge query) tells wither there is an edge between them. If the client wants to perform adjacency query for the pair of vertices v_1 and v_2, the client sends (i'_1, i'_2) (as generated in Algo. 28) to the CS. The CS returns $m_{i'_1 i'_2}$. The client can get either the randomized result from the PS or it can decrypt $m_{i'_1 i'_2}$ by itself.

7.5 Performance analysis

In this section, we discuss the efficiency of our proposed schemes. The efficiency is measured in terms of computations and communication complexities together with storage requirement and allowed leakages. A summary is given in Table 7.1. Since there is no work on the secure link prediction before, we have not compared complexities of our schemes with any other similar encrypted computations.

Table 7.1: Complexity Comparison Table

Param	Entity	SLP-I	SLP-II	SLP-III
Leakage	CS	$\|V\|, \tau_{v_1}, \tau_{v_2}, \ldots$	$\|V\|, \tau_{v_1}, \tau_{v_2}, \ldots$	$\|V\|, \tau_{v_1}, \tau_{v_2}, \ldots$
	PS	S_v, i_{res}	S'_v, i_{res}	i_{res}
Storage	client	λ bits	λ bits	λ bits
	CS	$\|V\|^2\rho$ bits	$2\|V\|^2\rho$ bits	$\|V\|^2\rho$ bits
	PS	ρ bits	ρ bits	ρ bits
Computation	client	$\|V\|^2(\mathsf{M}+\mathsf{A})$	$\|V\|^2(\mathsf{M}+\mathsf{A}+\mathsf{M_1}+\mathsf{A_1})$	$\|V\|^2(\mathsf{M}+\mathsf{A})$
	CS	$\|V\|^2$ P + $\|V\|$ E + $(\|V\|^2+\|V\|)$ M	$\|V\|^2$ P + $(\|V\|^2+2\|V\|)$ M	$\|V\|^2$ P + $4\|V\|$ E + $(\|V\|^2+3\|V\|)$ M + $MGC_{const}(\log\|V\|, \|V\|)$
	PS	$\|V\|log\|V\|(\mathsf{M}+\mathsf{C}+\mathsf{M_1}+\mathsf{C_1})$ + $\|V\|log\|V\|\mathsf{C}$	$\|V\|(\mathsf{M_1}+\mathsf{C_1})$ + + $\|V\|log\|V\|\mathsf{C}$	$\|V\|(\mathsf{M}+\mathsf{C}+\mathsf{M_1}+\mathsf{C_1})$+ $MGC_{eval}(\log\|V\|, \|V\|)$
Communication	client→CS	$\|V\|^2\rho$ bits	$2\|V\|^2\rho$ bits	$\|V\|^2\rho$ bits
	CS→PS	$2\|V\|\rho$ bits	$\|V\|\rho$ bits	$2\|V\|\rho$ bits + $\|V\|OT_{snd}^{(\log\|V\|+1)}$+ $MGC_{size}(\log\|V\|, \|V\|)$ bits
	PS→CS	-	-	$\|V\|OT_{rcv}^{(\log\|V\|+1)}$
	PS→client	$\log\|V\|$ bits	$2\|V\|\log\|V\|$ bits	$\log\|V\|$ bits

S_v - Set of scores of v with all other vertices, S'_v- a subset of S_v, ρ- length of elements in G or $\mathsf{G_1}$, C- comparison in G, $\mathsf{C_1}$- comparison in $\mathsf{G_1}$, M- multiplication in G, $\mathsf{M_1}$- multiplication in $\mathsf{G_1}$, E- exponentiation in G, $\mathsf{E_1}$- exponentiation in $\mathsf{G_1}$, P- pairing/ bilinear map computation, $MGC_{size}(\log|V|, |V|)$- size of MGC with $|V|\log|V|$-bit inputs, $MGC_{const}(\log|V|, |V|)$- MGC contraction with $|V|\log|V|$-bit inputs, $MGC_{eval}(\log|V|, |V|)$- MGC evaluation with $|V|\log|V|$-bit inputs, $OT_{snd}^{(\log|V|+1)}$- information to send for $(\log|V|+1)$-bit OT, $OT_{rcv}^{(\log|V|+1)}$- information to receive for $\log|V|$-bit OT.

7.5.1 Complexity analysis

Let the graph be $G = (V, E)$ and $N = |V|$. Let BGN encryption outputs ρ-bit string for every encryption. We describe the complexities as bellow.

Leakage Comparison: As we see the Table 7.1, each scheme leaks, to the CS, same amount of information which is the number of vertices of the graph and the query trapdoors. However, none of the schemes leaks information about the edges in the graph to the CS. In SLP-I, since the PS has the power to decrypt the scores, it gets to know $S_v = \{score(v, u) : u \in V\}$. However, SLP-II reveals only a subset S'_v of S_v and SLP-III manages to hide all scores from the PS. SLP-I cannot hide scores from the PS which results in maximum leakage to the PS.

Storage Requirement: One of the major goals of secure link prediction scheme is that the client should require very little storage. All our designed schemes have very low storage requirement for the client. The client has to only store a key which is of λ bits. For all schemes, the PS stores only a part of the secret key which is of λ bits.

In SLP-I, the CS is required to store $|V|^2\rho$ bits for the structure T where the PS is required to store only the secret key. While reducing the leakage in SLP-II, the CS storage becomes doubled. However, SLP-III requires the same amount of storage as SLP-I.

Computation Complexity: In all schemes, the client computes $|V|^2$ number of BGN encryption to encrypt A while SLP-II additionally computes $|V|^2$ number of the same to encrypt B. To compute each of $|V|$ encrypted scores, the CS requires $|V|$ bilinear map (e) computation and $|V|$ multiplications.

Additionally, SLP-I randomizes the encrypted entries corresponding to the row that has been queried. This requires $|V|$ exponentiations and $|V|$ multiplications. SLP-II randomizes the encrypted scores. This requires $|V|$ multiplications and computes the encrypted degree of the queried vertex which requires $|V|$ multiplications. Apart from computations of encrypted scores, in SLP-III, the CS computes a garbled circuit MGC.

In all, the PS decrypts $|V|$ scores. Each decryption requires $\log |V|$ multiplications on average. To find the vertex with maximum score, in SLP-I and SLP-II, the PS compares $|V|$ numbers. The $|V|$ encrypted entries are decrypted by the PS in SLP-I and SLP-III. In addition, the PS evaluates the garbled circuit MGC in SLP-III.

Communication Complexity: To upload the encrypted matrices, SLP-I and SLP-III requires $|V|^2\rho$ bits and SLP-II requires $2|V|^2\rho$ bits of communications. To query, it sends only the trapdoor

of size 2ρ bits (aprx.).

The CS sends $2|V|$ entries to the PS, in case of SLP-I and SLP-III. For SLP-II, the CS sends only $|V|$ entries. Each of these entries is of ρ bits. In addition, SLP-III sends the garbled circuit MGC. PS to CS communication happens only when the PS evaluates MGC. For SLP-I and SLP-III, the PS sends only i_{res} which is of $\log|V|$ bits to the client. However, the PS sends $2|V|\log|V|$ bits to the client.

Complexity for GC Computation: It can be observed that $\log|V|$-bit SUB, 1-bit SUB′, $\log|V|$-bit MUL, $\log|V|$-bit COMP and $\log|V|$-bit MUX blocks consist of ($4\log|V|$ XOR-gates and $\log|V|$ AND-gates), (4 XOR-gates and 1 AND-gate), ($\log|V|$ AND-gates), ($3\log|V|$ XOR-gates and $\log|V|$ AND-gates) and ($2\log|V|$ XOR-gates and $\log|V|$ AND-gates) respectively. Thus, $\log|V|$-bit NSS and $\log|V|$-bit Max blocks consist of ($(4\log|V|+4)$ XOR-gates and $(2\log|V|+1)$ AND-gates) and ($7\log|V|$ XOR-gates and $3\log|V|$ AND-gates) respectively.

In our designed garbled circuit MGC, there are $(|V|-1)$ Max blocks and $|V|$ NSS blocks. Thus, MGC requires $|V|(11\log|V|+4)$ XOR-gates and $|V|(5\log|V|+1)$ AND-gates. However, the PS receives $|V|(\log|V|+1)$ bits through OT for the first layer.

Thus, $MGC_{size}(\log|V|,|V|)$ is the size of $|V|(11\log|V|+4)$ XOR-gates and $|V|(5\log|V|+1)$ AND-gates, whereas $MGC_{const}(\log|V|,|V|)$ and $MGC_{eval}(\log|V|,|V|)$ are computational cost to construct and evaluate.

7.6 Experimental evaluation

In this section, the experimental evaluations of our designed schemes, SLP-I and SLP-II, are presented. In our experiment, we have used a single machine for both the client and the server. All data has been assumed to be residing in main memory. The machine is with an Intel Core i7-4770 CPU and with 8-core operating at 3.40GHz. It is equipped with 8GB RAM and runs an Ubuntu 16.04 LTS 64-bit operating system. The open source PBC [77] library has been used in our implementation to support BGN. The code is in the repository [80].

7.6.1 Datasets

For our experiment, we have used real-world datasets. We have taken the datasets from the *SNAP datasets* [55]. The collection consists of various kinds of real-world network data which includes social networks, citation networks, collaboration networks, web graphs etc.

Table 7.2: Detail of the graph datasets

Dataset Name	**#Nodes**	**#Edges**
bitcoin-alpha	3,783	24,186
ego-facebook	4,039	88,234
email-Enron	36,692	183,831
email-Eu-core	1,005	25,571
Wiki-Vote	7,115	103,689

For our experiment, we have considered the undirected graph datasets- *bitcoin-alpha*, *ego-Facebook*, *Email-Enron*, *email-Eu-core* and *Wiki-Vote*. The number of nodes and the edges of the graphs are shown in Table 7.2.

Instead of the above graphs, their subgraphs have been considered. First fixed number of vertices from the graph datasets and edges joining them have been chosen for the subgraphs. For example, for 1000, vertices with identifier < 1000 have been taken for the subgraph.

7.6.2 Experiment results

In our experiment, five datasets have been taken. The experiment has been done for each dataset taking extracted subgraphs with vertices 50 to 1000 incremented by 50. The number of edges in the subgraphs is shown in Fig. 7-5. For the pairing, 128, 256 and 512 bits prime-pairs are taken. In our proposed schemes, the most expensive operation for the client is encrypting the matrix

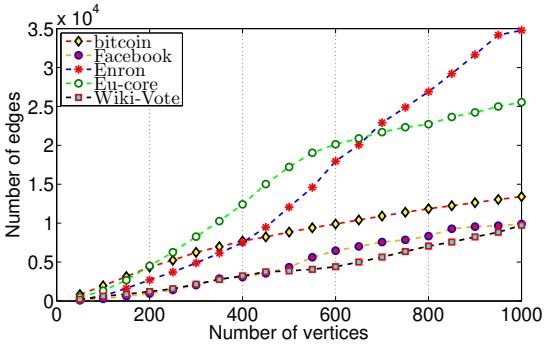
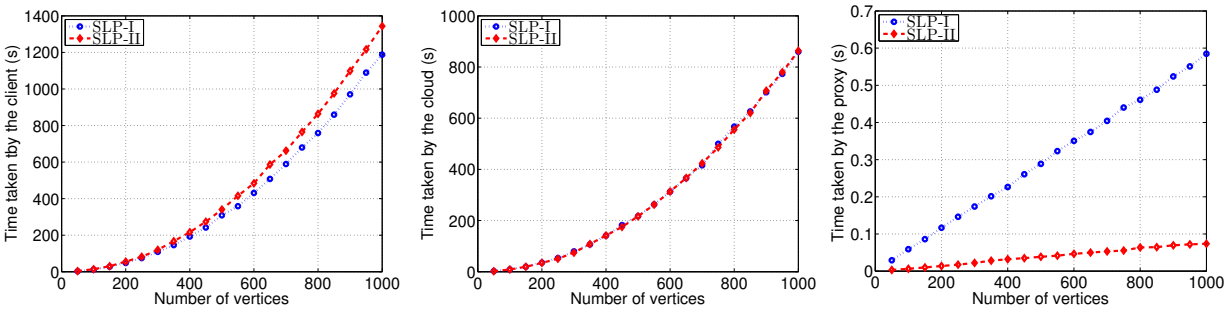


Figure 7-5: Number of vertices and edges of the subgraphs

(`EncMatrix`). For the cloud and the proxy, score computing (`LPQuery`) and finding maximum vertex (`FindMaxVertex`) are the most expensive operations respectively. Hence, throughout this section, we have discussed mainly these three operations.

As we have seen, in the proposed protocols, encrypting each entry of the adjacency matrix is the main operation of the encryption, the number of edges does not affect the encryption time for both SLP-I and SLP-II. This is because, irrespective of SLP schemes, the number of operations are independent of number of edges.



(a) Encryption time taken by the client (b) Encrypted score computation times (c) Score decryption and sorting times

Figure 7-6: comparison between SLP-I and SLP-II w.r.t. computation time when the primes are of 128 bits each

Similarly, time required by the cloud to compute score is independent of number of edges and depends on number of entries in the adjacency matrix i.e., N^2. Time taken for each of the operations is shown in Fig. 7-6. In the figure, we have compared time for both SLP-I and SLP-II taking primes 128 bits each.

However, the time taken by the proxy to decrypt the scores is depends on the number of vertices. In SLP-I, the proxy has to decrypt $|V|$ entries in G as well as $|V|$ scores in G_1 where in SLP-II, it decrypts only in $|V|$ scores in G_1. So proxy takes more time in SLP-I than in SLP-II. This can be observed in Fig. 7-6c.

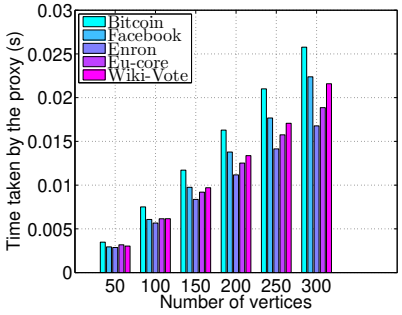
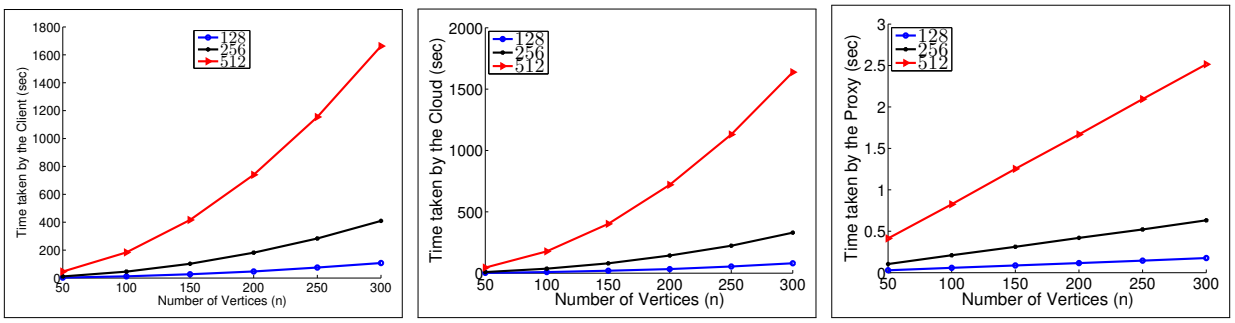


Figure 7-7: Time taken by the proxy in SLP-II for different datasets considering 128-bit primes

For a query, in SLP-II, the proxy decrypts scores only for corresponding vertices that are not incident to the vertex queried for. So, only in this case, the computational time depends on the

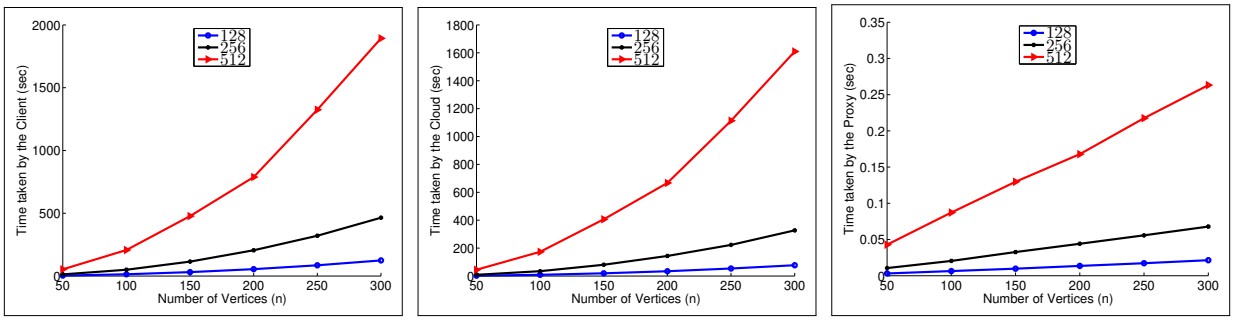
number of edges in the graph. As density of edges in a graph increases the chance of decreasing computational time for the graph increases. In Fig. 7-7 we have compared computational time taken by the proxy in SLP-II for different datasets.

In the above figures, we have considered only 128-bit primes. It can be observed from the experiment, the computational time depends on the security parameter. As we increase the size of the primes, the computational time grows exponentially. We have compared the change of computational time for all of the client, cloud and proxy for both SLP-I and SLP-II (see Fig. 7-8 and Fig. 7-9 respectively). However, in practical, as we keep the security bit fixed, keeping the security bits as low as possible improves the performance.



(a) Client time in SLP-I (b) Cloud time in SLP-I (c) Proxy time in SLP-I

Figure 7-8: Computational time in SLP-I with 128, 256 and 512-bit primes



(a) Client time in SLP-II (b) Cloud time in SLP-II (c) Proxy time in SLP-II

Figure 7-9: Computational time in SLP-II with 128, 256 and 512-bit primes

7.6.3 Estimation of computational cost in SLP-III

In the previous section, we have shown the experimental results for SLP-I and SLP-II. In this section, we have estimated the computational cost for SLP-III. Encryption algorithm of SLP-III

is same as SLP-I. So both required same amount of time for encryption for the same dataset. To estimate query time, we have considered a random graph with 10^3 vertices.

Query Time: In SLP-III the cloud computes encrypted scores and the proxy decrypts the scores as well as random numbers. The number of decryption in each group is same as SLP-I. However, in SLP-III, it requires an extra garbled circuit computation. For this, 1000 OT for 128-bit security of ECC is required which takes $138 * 1000$ms = 138s aprx. ([3, 70]). In addition to that, the PS evaluates the GC with $1000 * (11 * 257 + 4) = 2831000$ XOR-gates and $1000 * (5 * 257 + 1) = 1286000$ AND-gates. Assuming that the encryption used in each GC circuit is AES (128-bit), GC evaluation requires 2 AES decryption and the CS requires 8 encryption. As we see in [1], it requires 0.57 cycles per byte for AES. Thus, for evaluation in a single core processor, the PS requires (2*(1286000*256/8)*0.57) cycles = 46913280 cycles that takes $(46913280/(2.5 * 10^9)) = 0.019$s. Similarly, The CS requires 0.078s to construct the GC.

The estimated costs are measured with respect to a single core 2.5 GHz processor. However, in practice, the CS provides a large number of multi-core processors. As we see all the computations can be computed in *parallel*, the query cost can be reduced dramatically. Each of the above-mentioned costs can be improved to $\frac{cost}{p}$s with p processors and cost is $cost$.

7.7 Introduction to $\texttt{SLP}_k$

Let us define another variant of secure link prediction problem $\texttt{SLP}_k$. Instead of returning the vertex with highest score, an $\texttt{SLP}_k$ returns indices of k number of top-scored vertices.

Let, a graph $G = (V, E)$ is given. Then, the *top-k Link Prediction Problem* states that given a vertex $v \in V$, it returns a set of vertices $\{u_1, u_2, \ldots, u_k\}$ such that $score(v, u_i)$ is among top-k elements in S_v. The top-k link prediction scheme is said to be secure i.e., a secure top-k link prediction problem scheme ($\texttt{SLP}_k$) if, the servers do not get any meaningful information about G from its encryption or sequence of queries.

Our proposed schemes, SLP-I and SLP-II, can be extended to support $\texttt{SLP}_k$ queries. In SLP-I, the only change is that instead of returning only the index of the vertex with highest score, the proxy has to return the indices of the top-k highest scores to the client.

Conclusion

In this chapter, we have presented three schemes supporting secure link prediction query with different trade-off. In the next chapter, we study clustering coefficient computation on dynamic

data. We propose a novel scheme that allows the local clustering coefficient to be queried over the outsourced encrypted graphs.

Chapter 8

Securely Computing Clustering Coefficient for Outsourced Dynamic Encrypted Graph Data

In Chapter 7, we have seen that nowadays multiple communities are connected and their connection can be represented as graphs with nodes being entities (members) of the social community and edges representing the association between entities (members). We have also seen that Small and Medium-sized Enterprises (SMEs) outsource the graph data to some cloud servers due to its cost-effectiveness and security effectiveness [78]. Outsourcing them enables the enterprises not only easier to process requests and faster to respond but also increases the data survivable probability. Moreover, to protect from misuse of the data stolen by some unauthorized person, data needs to be encrypted before outsourcing [83, 84] without losing the ability to query. Since direct encryption prevents the cloud to perform queries, there is a need for a different encryption technique that allows queries to be performed on a controlled portion of the data.

In most real-world networks, nodes tend to create tightly knit groups characterized by a relatively high density of ties [98]. The nodes belonging to such a group have a higher probability to be connected in the future. The clustering coefficient is a measure of the degree to which nodes in a graph cluster or associate with one another [98]. It is an important measure metric to determine the structural properties of the graph. Like degree, betweenness, and centrality, it plays an important role to study the robustness of the network. Nodes with a high clustering coefficient are more prone to targeted attacks [53], hence need more protection. Similarly, networks with a high global clustering coefficient are less susceptible to targeted attacks.

Previously, in encrypted graphs, basic queries like vertex degree query, adjacency query, etc. have been studied [27]. If we consider complex queries, the link prediction has been studied in [83]. The shortest distance query, that returns shortest distance between two given points, has been studied in different ways in [84], [65], [97] etc. Xie and Xing [101] studied clustering coefficient on encrypted graph. However, they used a public-key encryption scheme, which makes the scheme inefficient for large datasets.

Keeping above all in mind, in this chapter, we have designed a novel graph encryption scheme Gopas that allows the local clustering coefficient to be queried over the outsourced encrypted graphs with update support. We have used only symmetric key encryption that makes the scheme Gopas efficient for large datasets. The scheme Gopas not only supports basic queries like edge query, vertex query, neighbor queries, etc., but it also allows the client to add new vertices and edges.

Our contribution In this chapter, we contribute to the following works.

1. We design a novel graph encryption scheme Gopas, called Dynamic clustering coefficient queryable Encryption ($DCCE$) scheme, supporting that performs clustering coefficient query on an outsourced encrypted appendable graph. The design is based on symmetric key encryption only. It allows performing neighbor queries as well as edge queries on the same encrypted graph. Moreover, it allows a new edge or vertex to be appended, making the scheme suitable for dynamic data.

2. We define the security of the clustering coefficient finding problem in the random oracle model. We show that the designed scheme Gopas is provably secure under the chosen-query attack.

3. We implement a prototype of the scheme Gopas and tested with multiple real-life *SNAP* [55] datasets. The implementation results show that the scheme Gopas is practical even for a very large database. For example, in a graph with 196,591 vertices and 950,327 edges, it takes only $1.2s$ approx for clustering coefficient query.

Organization We summarize our work in the chapter as follows. We discuss the system model, required cryptographic tools, etc. in Section 8.1. We present our proposed scheme, in Section 8.2, for static database and then we extend the scheme to support dynamic updates. In Section 8.3, we discuss the prototype of our scheme and its result.

8.1 Preliminaries

8.1.1 Clustering coefficient

Clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. These are of two types- local and global. Two versions of this measure exist; the global and

the local. The global version was designed to give an overall indication of the clustering in the network, whereas the local gives an indication of the embeddedness of single nodes. We define them as follows.

Let $G = (V, E)$ be an unweighted graph where V is the set of vertices and E is the set of edges between them. Let the edges between two vertices v_i and v_j is denoted by e_{ij} or v_iv_j. Let us consider N_{v_i} be the neighborhood of the vertex $v_i \in V$ and defined as $N_{v_i} = \{v_j : (e_{ij} \in E) \vee (e_{ji} \in E)\}$.

Definition 8.1 (Clustering Coefficient in Directed Graph). *[98] For the unweighted graph $G = (V, E)$, if the graph is directed,* local clustering coefficient *for a node $v_i \in V$ is denoted by $\hat{CC}_i$ and defined as*

$$\begin{aligned}\widetilde{CC}_i &= \frac{\text{\# pairs of closed neighbours of } v}{\text{\# pairs of neighbours of } v} \\ &= \frac{|\{e_{jk} : (v_j, v_k \in N_{v_i}) \wedge (e_{jk} \in E)\}|}{|N_{v_i}|(|N_{v_i}| - 1)}\end{aligned}$$

Definition 8.2 (Clustering Coefficient in Undirected Graph). *If the unweighted graph $G = (V, E)$ is undirected,* local clustering coefficient *for a node $v_i \in V$ is denoted by CC_i and defined as*

$$CC_i = \frac{|\{e_{jk} : (v_j, v_k \in N_{v_i}) \wedge (e_{jk} \in E)\}|}{|N_{v_i}|(|N_{v_i}| - 1)/2}$$

Let, a vertex v has neighborhoods $N_v = \{v_1, v_2, \ldots, v_{n_v}\}$. Let for each $v_i \in N_v$, $N_{v_i} = \{v_{i1}, v_{i2}, \ldots, v_{in_{v_i}}\}$ be the neighborhood of v_i. Then we can see that, if v_{ij} is in some triangle, then it must be present in N_v. So, the number of such closed triangles can be given as $|\{v_{ij} : v_{ij} \in N_v \setminus \{v_i\}\}|$.

Since, in the set each triangle is counted twice in an undirected graph, $|\{v_{ij} : v_{ij} \in N_v \setminus \{v_i\}\}|/2$ is the number of such triangles. Then the local clustering coefficient is given by $CC_i = \frac{|\{v_{ij}:v_{ij}\in N_v\setminus\{v_i\}\}|/2}{n_v(n_v-1)/2}$. Note that, for a complete graph $CC_i = 1$. It has been proved that in a classical random network $CC_i \to 0$.

Definition 8.3 (Global Clustering Coefficient). *In an unweighted graph $G = (V, E)$,* global clustering coefficient*, for a node $v_i \in V$, is $\frac{1}{|V|}\sum_{v\in V}\widetilde{CC}_v$, if G is directed and $\frac{1}{|V|}\sum_{v\in V} CC_v$, if G is undirecte*

In this chapter, we are interested in the local clustering coefficient.

8.1.2 System model

In our system model, there are three entities– owner, cloud, and user. They are shown in Figure 8-1 and briefly described as follows.

Owner is a *trusted* entity who owns the database. It generates the required keys and encrypts the graph data and uploads it to the cloud. It also has the ability to request queries.

Cloud is the cloud storage and computation service provider. It stores the encrypted graph data and performs a search over it on request from the user. It is considered to be *honest-but-curious* adversary who runs the protocol correctly but wants to learn information from the query.

User is an entity who wants to perform a query on the encrypted graph. It takes a token from the owner and then requests a search to the cloud.

Moreover, we assume that communication between the entities is done via secure channels.

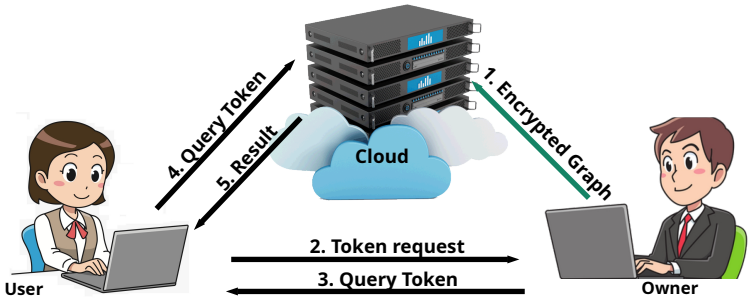


Figure 8-1: System model of the secure clustering coefficient computation

8.1.3 Design goals

Considering the above system model, we want to achieve confidentiality, scalability, efficiency, update support as described in Section 1.1.2. In addition, we want the followings.

Ability to query: In the design, *along with the clustering coefficient query*, the ability to perform the basic queries of a graph should be present. The basic queries include vertex query, edge query, and neighbor query. Given a vertex v_i, a *vertex query* returns whether $v_i \in V$. Given two vertices v_i and v_j, an *edge query* returns whether the edge e_{ij} present in the graph i.e., whether $e_{ij} \in E$. Given a vertex v_i, a *neighbor query* returns the set N_{v_i} of vertices adjacent to v_i.

Suitability: In the scheme, the user should do as less computation as possible to perform query so that it is suitable for lightweight devices. Moreover, the owner should require as less local storage as possible.

8.1.4 Definitions

Definition 8.4 (CCE)**.** *We define a* graph encryption scheme supporting clustering coefficient query *(CCE) scheme as a tuple of algorithms* $\Psi_s = (\texttt{KeyGen}, \texttt{Encrypt}, \texttt{CCQueryTkn}, \texttt{CCQuery})$ *described as follows.*

- $K \leftarrow \texttt{KeyGen}(\lambda)$: *Given a security parameter* λ*, the owner runs this probabilistic polynomial time (PPT) algorithm and outputs secret key* K *of the scheme.*
- $EG \leftarrow \texttt{Encrypt}(G, K)$*: This is a PPT algorithm run by the owner. Given the key* K *and the graph* G*, it outputs the encrypted graph* EG*.*
- $\tau^c \leftarrow \texttt{CCQueryTkn}(v, K)$*: This is a PPT algorithm, run by the owner. It outputs a token* τ_v^c *for clustering coefficient query after taking a vertex* v *and the key* K*.*
- $CC_v \leftarrow \texttt{CCQuery}(EG, \tau^c)$*: This is a cloud-side PPT algorithm that takes a query token* τ_v^c *and the encrypted graph* EG *as input and outputs the clustering coefficient* CC_v *of the vertex* v*.*

The above definition is for static database. We extend the definition to support updates as follows.

Definition 8.5 (Dynamic CCE)**.** *We define a* dynamic graph encryption scheme supporting clustering coefficient query (DCCE) *as a CCE scheme with additional update support. It has additional algorithms* `UpdtVertexTkn`*,* `UpdtVertex`*,* `UpdtEdgeTkn` *and* `UpdtEdge` *described as follows.*

- $\tau^v \leftarrow \texttt{UpdtVertexTkn}(v, K, op)$*: It is a owner-side PPT algorithm that takes the key* K*, a vertex* v *and an operation bit* op *and returns a vertex update token* τ^v*.*
- $EG' \leftarrow \texttt{UpdtVertex}(EG, \tau^v)$*: It is a cloud-side PPT algorithm that takes a token* τ^v*, and the encrypted graph* EG *and returns the updated encrypted graph* EG'*.*
- $\tau^e \leftarrow \texttt{UpdtEdgeTkn}(v_1, v_2, K, op)$*: It is a owner-side PPT algorithm that takes the key* K*, two vertices* v_1 *and* v_2 *and a bit* op *and returns an edge update token* τ^e*.*

- $EG' \leftarrow \texttt{UpdtEdge}(EG, \tau^e)$*: It is a cloud-side PPT algorithm that takes a token* τ^e*, and the encrypted graph* EG *and returns the updated encrypted graph* EG'*.*

Correctness A scheme $DCCE$ is said to be correct if $\forall \lambda \in N, \forall K$ generated using $KeyGen(1^\lambda)$ and all sequence of vertex and edge updates, every clustering coefficient query returns correct result except with a negligible probability.

8.1.5 Security

In our model, the cloud is considered to be the adversary $\mathcal{A}$ and the client is considered to be the challenger $\mathcal{C}$ which is simulated by a simulator $\mathcal{S}$. We define the security of a $DCCE$ is in real-ideal paradigm as follows.

Algorithm 36: $\textbf{Real}_{\mathcal{A}}^{\texttt{DCCE}}(\lambda)$

$K \leftarrow \texttt{KeyGen}(1^\lambda)$ [$\mathcal{C}$]
$(G, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$EG \leftarrow \texttt{Encrypt}(G, K)$
$(q_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, EG)$
for $1 \leq i \leq q$ **do**
    if q_i *is CC query for* v **then**
        $t_i \leftarrow \texttt{CCQueryTkn}(v, K)$
    else if q_i *is vetex update for* v *and* op **then**
        $t_i \leftarrow \texttt{UpdtVertexTkn}(v, K, op)$
    else if q_i *is edge update for* v_1, v_2 *and* op **then**
        $t_i \leftarrow \texttt{UpdtEdgeTkn}(v_1, v_2, K, op)$
    end
    if $i < q$ **then**
        $(q_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, EG, t_1, \ldots, t_{i-1})$
    end
end
$t = (t_1, t_2, \ldots, t_q)$
$b \leftarrow \mathcal{A}_{q+1}(EG, t, st_{\mathcal{A}})$, where $b \in \{0, 1\}$
return b

Definition 8.6 (Adaptive semantic security (CQA2) of a DCCE scheme)**.** *Let* DCCE *be a dynamic graph encryption scheme supporting as Definition 8.5. Let* $\mathcal{A}$ *be a stateful adversary,* $\mathcal{C}$ *be a challenger,* $\mathcal{S}$ *be a stateful simulator and* $\mathcal{L} = (\mathcal{L}_e, \mathcal{L}_c, \mathcal{L}_{vu}, \mathcal{L}_{eu})$ *be a stateful leakage algorithm. Let us consider two games-* $\textbf{\textit{Real}}_{\mathcal{A}}^{\texttt{DCCE}}(\lambda)$ *(see Algo. 36) and* $\textbf{\textit{Ideal}}_{\mathcal{A},\mathcal{S}}^{\texttt{DCCE}}(\lambda)$ *(see Algo. 37).*

Algorithm 37: Ideal$^{\texttt{DCCE}}_{\mathcal{A},\mathcal{S}}(\lambda)$

$(G, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^{\lambda})$
$(st_{\mathcal{S}}, EG) \leftarrow \mathcal{S}_0(\mathcal{L}_e(G))$
$(q_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, EG)$
for $1 \leq i \leq q$ **do**
  if q_i *is a CC query token* **then**
    $\mathcal{L}_i \leftarrow \mathcal{L}_c(q_1)$
  else if q_i *is vetex update for* v_1 *and* op **then**
    $\mathcal{L}_i \leftarrow \mathcal{L}_{vu}(v_1, op)$
  else if q_i *is edge update for* v_1, v_2 *and* op **then**
    $\mathcal{L}_i \leftarrow \mathcal{L}_{eu}(v_1, v_2, op)$
  end
  $(t_i, st_{\mathcal{S}}) \leftarrow \mathcal{S}_i(st_{\mathcal{S}}, \mathcal{L}_1, \ldots, \mathcal{L}_i)$
  if $i < q$ **then** $(q_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, EG, t_1, \ldots, t_{i-1})$
end
$t = (t_1, t_2, \ldots, t_q)$
$b' \leftarrow \mathcal{A}_{q+1}(EG, t, st_{\mathcal{A}})$, where $b' \in \{0, 1\}$
return b'

The DCCE *is said to be adaptively semantically $\mathcal{L}$-secure against chosen-query attacks (*CQA2*) if, $\forall$ PPT adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_{q+1})$, where $q = poly(\lambda)$, $\exists$ a PPT simulator $\mathcal{S} = (\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_q)$, such that*

$$|Pr[\textbf{\textit{Real}}^{\texttt{DCCE}}_{\mathcal{A}}(\lambda) = 1] - Pr[\textbf{\textit{Ideal}}^{\texttt{DCCE}}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda) \tag{8.1}$$

Thus, the security definition (Definition 8.6) ensures $\mathcal{A}$ cannot distinguish $\mathcal{C}$ from $\mathcal{S}$.

8.2 Our Proposed Protocol

In this section, we have proposed a secure *DCCE* scheme based on the hash table. The scheme securely computes local clustering coefficient in random undirected graphs.

8.2.1 Overview

For the graph $G = (V, E)$, two tables T_V and T_E (corresponding to V and E resp.) are maintained. There is an entry in T_V for each vertex $v \in V$, called *vertex-node*. It contains a left key l^o_v

and a right key r_v^o– initially both are kept encrypted. Moreover, it keeps auxiliary information a_v and n_{v^o}.

For every vertex v, its adjacency list is kept as a link list where list-nodes are stored in T_E and called *edge-nodes*. Each edge-node, corresponds to a adjacent vertex, say u (except the last node which is Θ), stores two information– 1. address n_u of the vertex-node and 2. its key l_u that can decrypt l_u^o, a_v and n_{v_o}. The address n_u and l_u are kept encrypted with l_u^o and l_u^o resp. The auxiliary information a_v and n_{v_o} helps to traverse through the adjacency list. The decryption key r_v to decrypt r_v^o can only be generated by the owner. Finally, the encrypted graph $EG = (T_V, T_E)$ is outsourced to the cloud.

To search for a vertex v, the user gives the cloud server (n_v, l_v, r_v). Cloud decrypts vertex-node at n_v completely with (l_v, r_v). It traverses its adjacency list with the help of a_v and n_{v_o} and decrypts them (edge-nodes) with fully with l_v^o, r_v^o. For an edge from v to u, the cloud then get n_u and left-key l_u. So, in the next level it can only get n_w for all vertices w adjacent to u and cannot go further levels.

Traversing so, the cloud can find $\{n_u, \{n_w : w \in N_u\} : u \in N_v\}\}$ from which the cloud can compute the clustering coefficient easily (see Section 8.1.1). We show an example of clustering coefficient computation for a small graph as follows.

Example: Let us take an example of a graph with 5 vertices $\{v_1, v_2, v_3, v_4, v_5\}$ with set of neighbors as $\{\{v_2, v_3, v_5\}, \{v_1, v_3, v_4, v_5\}, \{v_1, v_2\}, \{v_2\}, \{v_1, v_2\}\}$ (see Figure 8-2) respectively. The figure shows how the vertex-nodes (yellow) and edge-nodes (gray) are linked.

To search clustering coefficient for v_3 owner gives l_{v_3} and r_{v_3} to the cloud. The cloud gets $(a_{v_3}, l_{v_3}^o, n_{v_3}^o)$ and r_{v_3}. It gets $N_{v_3} = \{n_{v_1}, n_{v_2}\}$ with the help of $(a_{v_3}, n_{v_3}^o)$ and gets l_{v_1} and l_{v_2} with the help of r_{v_3}. Then it goes to vertex node n_{v_1} and finds similarly $N_{v_1} = \{n_{v_2}, n_{v_3}, n_{v_5}\}$. Since, $n_{v_2} \in N_{v_1}$ and $n_{v_2} \in N_{v_3} \setminus \{n_{v_1}\}$ the triangle counts to 1. Similarly, the cloud gets $N_{v_2} = \{n_{v_1}, n_{v_3}, n_{v_4}, n_{v_5}\}$ and gets 1 more triangle. Thus the cloud returns local clustering coefficient $\frac{2}{2(2-1)} = 1$.

8.2.2 Scheme description

Here we propose a $DCCE$ scheme Ψ_s, suitable for the static database. It is divided into three phases– key generation, encryption, and query phases. We describe them as follows.

Key generation phase: In this phase, given the security parameter be λ, the owner first generates key $K = (k_n, k_a, k_l, k_r, k_l^o, k_r^o, k_s)$ for the scheme, each of which, except k_n, is a λ-bit string

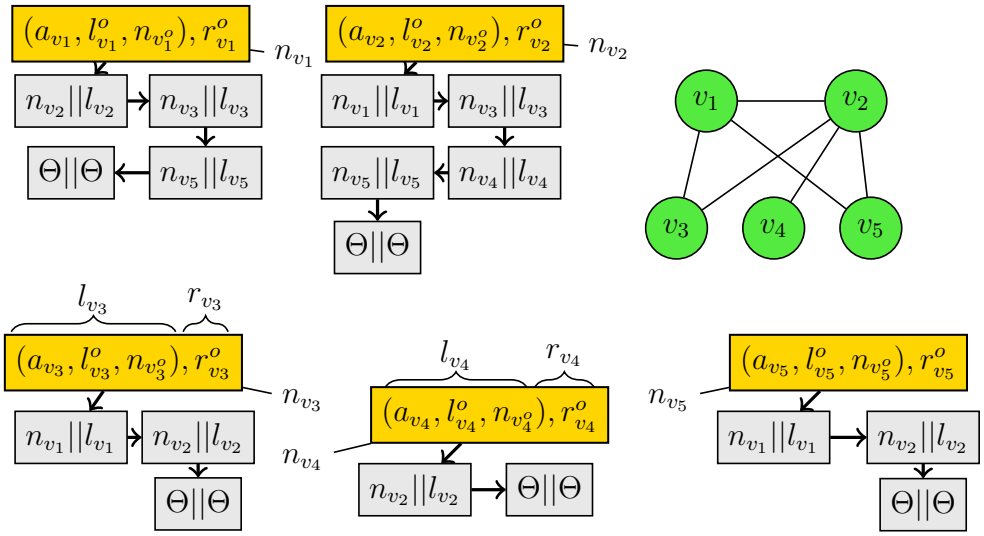


Figure 8-2: An example of what nodes store (before encryption)

chosen at random. The part k_n is generated as $\mathtt{SE}.\mathtt{Gen}(1^\lambda)$ where $\mathtt{SE} = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ is a symmetric key encryption scheme.

Encryption phase: In this phase, the owner takes a graph $G = (V, E)$ and encrypts it to EG with the key K as given in Algo 38. At first, the owner takes two hash tables T_V and T_E corresponding to the set V and E respectively. Since both T_V and T_E are hash tables, it stores key-value pairs.

It takes a PRP $P : \{0,1\}^\lambda \times \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$, a PRG $F : \{0,1\}^\lambda \rightarrow \{0,1\}^{3\lambda}$, and a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$.

From the top, the owner stores each vertex and their adjacency lists in them. For a vertex $v \in V$, its encrypted identifier n_v of v is used as the key while storing it in T_V and the value is the encrypted (with $F(l_v)$ and r_v) information of its adjacency list. The value also stores information (l^o_v, r^o_v) to decrypt the list. As the information of the adjacency list, the encrypted address n_{v_o} of the first node of the list is stored together with a_v that helps to find the address of the nodes in the list.

The adjacency list is stored as a linked list. Each entry of the list stores n_{v_i} that helps to find the address of the next entry. It stores an additional information l_{v_i} that can decrypt the entry corresponding to v_i in T_V partially. Each entry of the adjacency list corresponds to an edge and is stored in T_E.

Note that, the last neighbor of a vertex is stored in the list S_V hidden with $H(k_s, n_v)$ in the position g_v. A hash function G is used to find the position in the table S_V. However, this table S_V is required only when the data is dynamic.

Algorithm 38: $\texttt{Encrypt}(G = (V, E), K)$

$(k_n, k_a, k_l, k_r, k_l^o, k_r^o, k_s) \leftarrow K$
Initialize empty hash table T_V and T_E.
for $v \in V$ **do**
  $l_v \leftarrow P(k_l, v)$; $r_v \leftarrow P(k_r, v)$
  $l_v^o \leftarrow P(k_l^o, v)$; $r_v^o \leftarrow P(k_r^o, v)$
  $n_v \leftarrow \texttt{SE.Enc}(k_n, v)$; $a_v \leftarrow P(k_a, v)$
  $key \leftarrow n_v$; $n_{v_o} \xleftarrow{\$} \{0,1\}^\lambda$
  $val \leftarrow ((a_v, l_v^o, n_{v_o}) \oplus F(l_v), r_v^o \oplus r_v)$
  $T_V[key] \leftarrow val$; $d \leftarrow deg(v)$
  for $i = 1$ **to** $i = d + 1$ **do**
    if $i > d$ **then**
      $n_{v_i} \leftarrow null$; $l_{v_i} \leftarrow null$; $\boxed{n'_{v_d} = n_{v_i};}$
    else
      $n_{v_i} \leftarrow \texttt{SE.Enc}(k_n, v_i)$; $l_{v_i} \leftarrow P(k_l, v_i)$
    end
    $key^o \leftarrow H(a_v, n_{v_{i-1}})$
    $val^o \leftarrow (n_{v_i} \oplus H(l_v^o, n_{v_{i-1}}), l_{v_i} \oplus H(r_v^o, n_{v_{i-1}}))$
    $T_E[key^o] \leftarrow val^o$
  end
  $g_v = G(v)$; $\boxed{S_V[g_v] \leftarrow n'_{v_d} \oplus H(k_s, n_v);}$
end
return $EG = (T_V, T_E)$; $\boxed{EG = (T_V, T_E, S_V)}$

Query phase: As we see in Section 8.1.1, to clustering coefficient of a vertex v, it is enough to find neighbors of neighbors of v. So, we first find a way to find neighbors of a vertex.

Neighbor Query: Each entry, in each adjacency list L_v of v, is stored in T_E and has two parts– encrypted neighbor vertex and an additional decryption information. To find adjacency list, only the first information is enough. It can be computed only if l_v $(= P(k_l, v))$, which is stored in n_v $(= \texttt{SE.Enc}(k_n, v))$. So, token $\tau_n = (n_v, l_v)$ is returned to search neighbors of v.

Algorithm 39: $\texttt{NeighborQuery}(T_E, T_V, \tau_n)$

$(n_v, l_v) \leftarrow \tau_n$
$val \leftarrow T_V[n_v]$
if $val = \perp$ **then**
  return ϕ
end
$(a_v, l_v^o, n_{v_o}) \leftarrow val[0] \oplus F(l_v)$; $i = 0$
while *1* **do**
  $key^o \leftarrow H(a_v, n_{v_{i-1}})$
  $val^o \leftarrow T_E[key^o]$
  $n_{v_i} \leftarrow val^o[0] \oplus H(l_v^o, n_{v_{i-1}})$
  if $(n_{v_i} = null)$ **then**
    break;
  else
    $i = i + 1$;
  end
end
return $\{n_{v_1}, n_{v_2}, \ldots, n_{v_i}\}$

To search the set of neighbors (see Algo 39), the cloud, at first, decrypts the left-right parts of the corresponding node in T_V. This gives (a_v, l_v^o, n_{v_o}) which helps to find the address of the neighbors in T_E. It can be seen that the right parts of the entries in T_E cannot be decrypted and hence cloud cannot go to the next level.

Clustering Coefficient Query: The main difference of the CC query from the neighbor query is that, in the CC query, the entry in T_V corresponding to the queried vertex is completely decrypted. So, given a vertex v, a token $\tau_c = (n_v, l_v, r_v)$ is returned where $n_v \leftarrow \texttt{SE.Enc}(k_n, v)$, $l_v \leftarrow P(k_l, v)$ and $r_v \leftarrow P(k_r, v)$. The owner generates the token and gives it to the user. The user sends it to the cloud whenever requires. r_v allows to decrypt r_v^o (see Algo 40) that is used to decrypt right parts of the neighbors stored in T_E. In the first round, the cloud computes neighbors of v completely. In the next round, for each neighbor v_i, the cloud performs neighbor queries which allows it to

Algorithm 40: `CCQuery`$(EG = (T_E, T_V), \tau_c)$

$(n_v, l_v, r_v) \leftarrow \tau_c$
$val \leftarrow T_V[n_v]$
if $val = \perp$ **then**
  return ϕ
end
$(a_v, l_v^o, n_{v_0}) \leftarrow val[0] \oplus F(l_v)$
$r_v^o \leftarrow val[1] \oplus r_v$
$i = 0$
while *1* **do**
  $key^o \leftarrow H(a_v, n_{v_{i-1}})$
  $val^o \leftarrow T_E[key^o]$
  $n_{v_i} \leftarrow val^o[0] \oplus H(l_v^o, n_{v_{i-1}})$
  $l_{v_i} \leftarrow val^o[1] \oplus H(r_v^o, n_{v_{i-1}})$
  if $((n_{v_i} = null) \wedge (l_{v_i} = null))$ **then**
    break
  else
    $i = i + 1$
  end
end
$R_0 \leftarrow \{n_{v_1}, n_{v_2}, \ldots, n_{v_i}\}$
$d = i$; $s = 0$
for $i = 1$ **to** $i = d$ **do**
  $\tau_n \leftarrow (n_{v_i}, l_{v_i})$
  $R_i \leftarrow$ `NeighborQuery`(T_E, T_V, τ_n)
  $s \leftarrow s + |R_0 \cap R_i|$
end
$cc \leftarrow \frac{s}{d(d-1)}$
return cc

decrypt only the right parts of them. These prevent the cloud to go further levels. Finally, the cloud matches the set of neighbors of v with the neighbors of neighbors and outputs the desired result.

Additional Queries: Though we have shown how to find neighbors, we see that checking whether a vertex or edge exists or not is a basic operation of a graph. Our proposed scheme Ψ_s allows both with constant time. To check whether a vertex v is present or not, it is enough to check whether entry in T_V, corresponding to key $n_v = \texttt{SE.Enc}(k_n, v)$, is present or not. To check whether an edge uv exists, it is enough to check whether the corresponding entry $H(a_u, n_v)$ is present in T_E or not, where $a_u = P(k_a, u)$ and $n_v = \texttt{SE.Enc}(k_n, v)$.

8.2.3 Computing Clustering Coefficient in Dynamic graphs

We extend the CCE scheme Ω_s to $DCCE$ scheme Ω_d that have additional update support. Updating a graph means mainly updating some vertex or updating some edge. In this section, we show how we add a new vertex or edge.

In Ω_d, key generation and query algorithms are the same as Ω_s. In `Encrypt`, corresponding to each vertex v, the owner just stores the last added neighbor n_{v_d} in a hash table S_V as $S_V[g(v)] \leftarrow n_{v_d}$, where g is a one-way hash function only computed by the owner. The table S_V it outsourced to the cloud. The additional steps are shown in boxes in Algorithm 38.

Adding a vertex: To add a vertex v, the owner generates a corresponding entry in T_V which is a key-value pair (key, val) (see Algo 41). It initializes an adjacency list with key-value pair (key^o, val^o) for T_E. Finally, the pair $(g(v), s_v)$ is computed to add it in S_V. Then the pairs are returned as token. The cloud adds them to their respective table when it receives the token (see Algo 42). We see that adding a vertex is a half-round process where the owner just sends a set of key-value pairs.

Adding an edge: (See Algo 43 and Algo 44). Since we have considered the graph to be undirected, adding an edge u-v means adding both the edges u-v and v-u. To add an edge u-v, we have to append a pair (key_u, val_u) (See Algo 43) in the adjacency list of u in T_E. Since the list is a link list, the last link should be updated. The update is done by the pair $(key_{u'}, val_{u'})$. Similarly, it is done with vertex v with the pairs (key_v, val_v) and $(key_{v'}, val_{v'})$. The owner computes and sends the pairs as token τ_{ae}. While computing the pairs, owner requires an extra round of communication to access the list S_V. Finally, all the four pairs are added in T_E (see Algo 44) by the cloud.

Algorithm 41: $\texttt{AddVertexToken}(K, v)$

$(k_n, k_a, k_l, k_r, k_l^o, k_r^o, k_s) \leftarrow K$
$n_v \leftarrow \texttt{SE.Enc}(k_n, v)$; $a_v \leftarrow P(k_a, v)$; $n_{v_o} \xleftarrow{\$} \{0,1\}^\lambda$
$l_v \leftarrow P(k_l, v)$; $r_v \leftarrow P(k_r, v)$;
$l_v^o \leftarrow P(k_l^o, v)$; $r_v^o \leftarrow P(k_r^o, v)$; $g_v \leftarrow g(v)$
$key \leftarrow n_v$; $val \leftarrow ((a_v, l_v^o, n_{v_o}) \oplus F(l_v), r_v^o \oplus r_v)$
$key^o \leftarrow H(a_v, n_{v_0})$;
$val^o \leftarrow (null \oplus H(l_v^o, n_{v_0}), null \oplus H(r_v^o, n_{v_0}))$
$g_v \leftarrow G(v)$; $s_v \leftarrow n_{v_0} \oplus H(k_s, n_v)$;
return $\tau_{av} = [(key, val), (key^o, val^o), g_v, s_v]$

Algorithm 42: $\texttt{AddVertex}(EG = (T_E, T_V), \tau_{av})$

$[(key, val), (key^o, val^o), g_v, n_{v_0}] \leftarrow \tau_{av}$
$T_V[key] \leftarrow val$; $T_E[key^o] \leftarrow val^o$; $S_V[g_v] \leftarrow s_v$
return

Algorithm 43: $\texttt{AddEdgeToken}(K, u, v)$

$(k_n, k_a, k_l, k_r, k_l^o, k_r^o, k_s) \leftarrow K$
$a_u \leftarrow P(k_a, u)$; $a_v \leftarrow P(k_a, v)$
$r_u^o \leftarrow P(k_r^o, u)$; $l_u^o \leftarrow P(k_l^o, u)$
$r_v^o \leftarrow P(k_r^o, v)$; $l_v^o \leftarrow P(k_l^o, v)$
$l_u \leftarrow P(k_l, u)$; $l_v \leftarrow P(k_l, v)$
$n_u \leftarrow \texttt{SE.Enc}(k_n, u)$; $n_v \leftarrow \texttt{SE.Enc}(k_n, v)$
$g_u \leftarrow G(u)$; $g_v \leftarrow G(v)$
$s_u \leftarrow S_V[g_u]$; $s_v \leftarrow S_V[g_v]$
$n_{u'} \leftarrow s_u \oplus H(k_s, n_u)$; $n_{v'} \leftarrow s_v \oplus H(k_s, n_v)$
$key_u \leftarrow H(a_u, n_v)$; $key_v \leftarrow H(a_v, n_u)$
$key_{u'} \leftarrow H(a_u, n_{u'})$; $key_{v'} \leftarrow H(a_v, n_{v'})$
$val_u \leftarrow (null \oplus H(l_u^o, n_v), null \oplus H(r_u^o, n_v))$
$val_v \leftarrow (null \oplus H(l_v^o, n_u), null \oplus H(r_v^o, n_u))$
$val_{u'} \leftarrow (n_v \oplus null, l_u \oplus null)$
$val_{v'} \leftarrow (n_u \oplus null, l_v \oplus null)$
$s'_u = s_u \oplus n_{u'} \oplus n_v$; $s'_v = s_v \oplus n_{v'} \oplus n_u$
return $\tau_{ae} = [(key_u, val_u), (key_{u'}, val_{u'}), (key_v, val_v), (key_{v'}, val_{v'}), (g_u, s'_u), (g_v, s'_v)]$

Correctness: The correctness of the scheme follows from the construction and the pseudo-randomness of the function P and H for the table T_V and T_E respectively. The scheme returns the correct result only if P or H returns unique identifiers for each unique vertex or unique edge. The uniqueness

Algorithm 44: `AddEdge`(T_E, τ_{ae})

$[(key_u, val_u), (key_{u'}, val_{u'}), (key_v, val_v), (key_{v'}, val_{v'}), (g_u, s'_u), (g_v, s'_v)] \leftarrow \tau_{add}$
$T_E[key_{u'}] \leftarrow T_E[key_{u'}] \oplus val_{u'}$; $T_E[key_u] \leftarrow val_u$
$T_E[key_{v'}] \leftarrow T_E[key_{v'}] \oplus val_{v'}$; $T_E[key_v] \leftarrow val_v$
$S_V[g_u] \leftarrow s'_u$; $S_V[g_v] \leftarrow s'_v$
return

follows from the property of a dynamic hash table.

8.2.4 Security analysis

Leakage due to initial dictionary construction is $L_e(G) = \{|T_E|, \{n_v : v \in V\}, \{s_v : v \in V\}\}$, due to neighbor query is $\mathcal{L}_n(v) = \{n_v, d_v, \{n_{v_i} : i = 0, \ldots, d_v\}\}$. The leakage in a clustering coefficient query is $\mathcal{L}_c(v) = \{n_v, d_v, \{(n_{v_i}, d_{v_i}, \mathcal{L}_n(v_i)) : v_i \in N_v\}\}$. The leakage due to add a vertex and add an edge are $\mathcal{L}_{av}(v) = \{key, key^o, g_v\}$ and $\mathcal{L}_{ae}(u, v) = \{key_u, key'_u, g_u, key_v, key'_v, g_v, \mathcal{L}_n(u), \mathcal{L}_n(v)\}$ respectively. Thus we have the leakage function $\mathcal{L} = \{\mathcal{L}_e, \mathcal{L}_n, \mathcal{L}_c, \mathcal{L}_{ae}, \mathcal{L}_{av}\}$.

Theorem 8.1. *If F, P, G, H and* `SE` *is a PRG, a PRP, a PRP, a hash functions and a CPA-secure symmetric key encryption respectively, then* `DCCE` *is $\mathcal{L}$-secure against adaptive dynamic chosen-query attacks, in random oracle model.*

Proof. According to the Definition 8.6, for any PPT adversary $\mathcal{A}$, it is sufficient to show that, $\exists$ a simulator $\mathcal{S}$ which cannot distinguish the output of $\mathbf{Real}_{\mathcal{A}}^{\texttt{DCCE}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\texttt{DCCE}}(\lambda)$ computationally. To show them indistinguishable, it is enough to show that so is the query tokens in the two worlds. We construct such a simulator $\mathcal{S}$ as follows.

Simulating H: Given a pair (x, y) of λ-bit strings, it finds $r_s \leftarrow RO_H[(x, y)]$. If $r_s \neq \perp$, then it returns r_s. Otherwise, it takes a λ-bit random λ-bit string $r_p \xleftarrow{\$} \{0,1\}^\lambda$, save it in the table as $RO_H[(x, y)] \leftarrow r_s$ and return r_s.

Simulating F: Given a λ-bit string x, it returns $r_s \leftarrow RO_F[(x)]$, in case $r_s \neq \perp$. Otherwise, it selects a random 3λ-bit string r_s, save it in the table as $RO_F[(x)] \leftarrow r_s$ and finally returns r_s.

Simulating P: The PRP P is used in $DCCE$ with 5 different keys, which are fixed throughout. So, $\mathcal{S}$ takes a table RO_P that stores 5-tuples (a, r, l, r^o, l^o) of λ-bit strings, for every λ bit input x. Given x, it finds $r_s \leftarrow RO_P[x]$. If $r_s \neq \perp$, then it returns $r_s = (a, r, l, r^o, l^o)$. Otherwise, it takes a 5λ-bit random number $r_s \xleftarrow{\$} \{0,1\}^{5\lambda}$, save it in the table as $RO_P[x] \leftarrow r_s$ and return r_s. Note that, the elements in the tuple corresponds to the key $k_a, k_r, k_l, k_r^o, , k_l^o$.

Simulating Encryption: Given the leakage $L_e(G) = \{|T_E|, \{n_v : v \in V\}, \{s_v : v \in V\}\}$, $\mathcal{S}$ builds simulated tables $\tilde{T_V}$, $\tilde{T_V}$ and $\tilde{S_V}$. $\mathcal{S}$ takes a one-way bijective map $M : \{0,1\}^\lambda \to \{0,1\}^\lambda$ that maps each λ-bit string n_v to an another λ-bit string $\tilde{n}_v = \texttt{SE.Enc}(\tilde{k}_n, 0^\lambda)$, where $\tilde{k}_n$ is randomly taken. Since the keys in table T_V are n_v, $\mathcal{S}$ keeps $\tilde{n}_v = M(n_v)$ as the corresponding key in $\tilde{T_V}$. Each value in T_V is a tuple of 4 completely random λ-bit strings. Thus for each n_v, $\tilde{T_V}[\tilde{n}_v] \leftarrow (bs_1, bs_2, bs_3, bs_4)$ where each bit string bs_i is a λ-bit string taken at random.

$\mathcal{S}$ takes a bijective map M' that maps each element g_v to $\tilde{g_v}$ i.e., $M'(g_v) = \tilde{g}_v$. Then it takes a table $\tilde{S_V}$ of size $|V|$ and stores random λ-bit string $\tilde{s}_v$ corresponding to $\tilde{g}_v$ as $\tilde{S_V}[\tilde{g}_v] = \tilde{s}_v$. It takes the table $\tilde{T_E}$ to simulate T_E, having same length. $\tilde{T_E}$ consists of $|T_E|$ key-value pairs where keys are λ-bit strings whereas values are 2λ-bit strings. The strings are chosen at random.

Simulating neighbor query: Given the leakage $\mathcal{L}_n(v) = \{n_v, d_v, \{n_{v_i} : i = 0, \ldots, d_v\}\}$, $\mathcal{S}$ finds $\tilde{n}_v \leftarrow M(n)$ and $\tilde{n}_{v_i} \leftarrow M(n_{v_i}), \forall i = 0, \ldots, d_v$. It first finds $(\tilde{a}_v, \tilde{r}_v, \tilde{l}_v, \tilde{r}^o_v, \tilde{l}^o_v) \leftarrow RO_P[\tilde{n}_v]$. If n_v is searched before by either neighbor query or cc query, then it returns $\tilde{\tau}_n = (\tilde{n}_v, \tilde{l}_v)$. As earlier if RO_P returns $\perp$, then a random string is appended in RO_P and returns the same. Then it assigns $RO_F[\tilde{l}_v] \leftarrow val[0] \oplus (\tilde{a}_v, \tilde{l}_v, \tilde{n}_{v_o})$ where $val = \tilde{T_V}[\tilde{n}_v]$.

Now, for each $i = 1$ to $i = d_v$, if $RO_H[\tilde{a}_v, \tilde{n}_{v_{i-1}}]$ is not searched before, then $\mathcal{S}$ takes an unused pair (key^o, val^o) from $\tilde{T_E}$, and updates RO_H as $RO_H[(\tilde{a}_v, \tilde{n}_{v_{i-1}})] = key^o$ and $RO_H[(\tilde{l}^o_v, \tilde{n}_{v_{i-1}})] = val^o[0] \oplus \tilde{n}_{v_i}$. Then, $\mathcal{S}$ assigns $RO_H[(\tilde{a}_v, \tilde{n}_{v_{d_v}})] = key^o$ and $RO_H[(\tilde{l}^o_v, \tilde{n}_{v_{d_v}})] = val^o[0] \oplus null$. Finally, $\mathcal{S}$ returns the simulated token $\tilde{\tau}_n = (\tilde{n}_v, \tilde{l}_v)$.

Simulating CC query: Given leakage $\mathcal{L}_c = \{n_v, d_v, \{(n_{v_i}, d_{v_i}, \mathcal{L}_n(v_i)) : v_i \in N_v\}\}$, it computes $\tilde{n}_v \leftarrow M(n_v)$. and then finds $(\tilde{a}_v, \tilde{r}_v, \tilde{l}_v, \tilde{r}^o_v, \tilde{l}^o_v) \leftarrow RO_P[\tilde{n}_v]$. If n_v is searched before either in clustering query or neighbor query, it finds $val \leftarrow \tilde{T_V}[\tilde{n}_v]$ and set $RO_F[\tilde{l}_v] \leftarrow val[0] \oplus (\tilde{a}_v, \tilde{l}_v, \tilde{n}_{v_o})$. If n_v is searched before either in clustering query, the token $\tilde{\tau}_c = (\tilde{n}_v, \tilde{r}_v, \tilde{l}_v)$ is returned. If n_v is not searched before in clustering query, then it does the followings.

1. $\forall i = 0, \ldots, d_v$, compute $\tilde{n}_{v_i} \leftarrow M(n_{v_i})$
2. For $\forall i = 1, \ldots, d_v$, if $RO_H[(\tilde{a}_v, \tilde{n}_{v_i})]$ is searched before then returns the corresponding (key^o, val^o) pair from $\tilde{T_E}$ where $key^o = RO_H[(\tilde{a}_v, \tilde{n}_{v_i})]$, else it takes a random unused pair (key^o, val^o) from from $\tilde{T_E}$. Finally, it assigns $RO_H[(\tilde{a}_v, \tilde{n}_{v_i})] = key^o$, $RO_H[(\tilde{l}_v, \tilde{n}_{v_i})] \leftarrow val^o[0] \oplus \tilde{n}_{v_{i+1}}$, and $RO_H[(\tilde{r}_v, \tilde{n}_{v_i})] \leftarrow val^o[1] \oplus \tilde{l}_{v_{i+1}}$. Here, $\tilde{n}_{v_{d_v}} = null$ and $\tilde{l}_{v_{d_v}} = null$
3. For $\forall i = 1, \ldots, d_v$, S simulates neighbor queries for the leakages as $\mathcal{L}_n(v_i)$.

Simulating add vertex query: Given leakage $\mathcal{L}_{av}(v) = \{key, key^o, g_v\}$, $\mathcal{S}$ computes $\tilde{n}_v \leftarrow M(n_v)$. It selects 5 λ-bit strings $(\tilde{a}_v, \tilde{r}_v, \tilde{l}_v, \tilde{r}^o_v, \tilde{l}^o_v)$ and append it in RO_P as $RO_P[\tilde{n}_v] = (\tilde{a}_v, \tilde{r}_v, \tilde{l}_v, \tilde{r}^o_v, \tilde{l}^o_v)$.

The it takes another random string $\tilde{n}_{v_0} \xleftarrow{\$} \{0,1\}^\lambda$
$val = ((\tilde{a}_v, \tilde{r}_{v_0}, \tilde{n}_{v_0}) \oplus RO_F[\tilde{r}_v], \tilde{l}_v^o \oplus \tilde{l}_v)$
$RO_F[\tilde{r}_v] \leftarrow val[0] \oplus (\tilde{a}_v, \tilde{r}_{v_0}, \tilde{n}_{v_0})$; $RO_H[(\tilde{a}_v, \tilde{n}_{v_0})] = key^0$
$RO_H[(\tilde{l}_v, \tilde{n}_{v_0})] \leftarrow val^o[0] \oplus null$
$RO_H[(\tilde{r}_v, \tilde{n}_{v_0})] \leftarrow val^o[1] \oplus null$
$\tilde{g}_v = M'(g_v)$; $\tilde{s}_v = \tilde{S}_V[\tilde{g}_v]$; $RO_{H_s}(\tilde{n}_v) = \tilde{s}_v \oplus \tilde{n_v}$
Finally, $\mathcal{S}$ returns $\tilde{\tau}_{av} = [(\widetilde{key}, \widetilde{val}), (\widetilde{key}^o, \widetilde{val}^o), \widetilde{g}_v, \widetilde{n}_{v_0}]$

Simulating add edge query: Given the leakage $\mathcal{L}_{ae}(u,v) = \{key_u, key'_u, g_u, key_v, key'_v, g_v, \mathcal{L}_n(u), \mathcal{L}_n(v)\}$, $\mathcal{S}$ does the followings.

1. $n_v = val_{u'} \oplus null$; $n_u = val_{v'} \oplus null$
2. $\tilde{n}_u = M(n_u)$; $\tilde{n}_v = M(n_v)$
3. $(\tilde{a}_u, \tilde{r}_u, \tilde{l}_u, \tilde{r}_u^o, \tilde{l}_u^o) \leftarrow RO_P[\tilde{n_u}]$
4. $(\tilde{a}_v, \tilde{r}_v, \tilde{l}_v, \tilde{r}_v^o, \tilde{l}_v^o) \leftarrow RO_P[\tilde{n_v}]$
5. $\widetilde{key}_u \leftarrow RO_H[(\tilde{a}_u, \tilde{n}_v)]$; $\widetilde{key}_v \leftarrow RO_H(\tilde{a}_v, \tilde{n}_u)$
6. $\widetilde{val}_u \leftarrow ((RO_H[(\tilde{l}_u, \tilde{n}_v)]), RO_H[(\tilde{r}_u, \tilde{n}_v)])$
7. $\widetilde{val}_v \leftarrow ((RO_H[(\tilde{l}_v, \tilde{n}_u)]), RO_H[(\tilde{r}_v, \tilde{n}_u)])$
8. $RO_H[(\tilde{l}_u, \tilde{n}_v)] = \widetilde{val}_u \oplus null$
9. $RO_H[(\tilde{r}_u, \tilde{n}_v)] = \widetilde{val}_v \oplus null$
10. $\tilde{T}_E[\widetilde{key}_u] = \widetilde{val}_u$; $\tilde{T}_E[\widetilde{key}_v] = \widetilde{val}_v$
11. Simulate neighbor queries for n_u and n_v, from which $n_{u'}$ and $n_{v'}$ can be found.
12. $\widetilde{key}_{u'} \leftarrow RO_H[(\tilde{a}_u, \tilde{n}_{u'})]$; $\widetilde{key}_{v'} \leftarrow RO_H[(\tilde{a}_v, \tilde{n}_{v'}])$
13. $\tilde{g}_u = M'(g_u)$; $\tilde{g}_v = M'(g_v)$
14. $\tilde{s}_u = \tilde{S}_V[\tilde{g}_u]$; $\tilde{s}_v = \tilde{S}_V[\tilde{g}_v]$
15. If n_u is appearing first time either in add edge or add vertex or neighbor query then $RO_{H_s}(\tilde{n}_u) = \tilde{s}_u \oplus \tilde{n_{u'}}$

16. If n_v is appearing first time either in add edge or add vertex or neighbor query then $RO_{H_s}(\tilde{n}_v) = \tilde{s}_v \oplus \tilde{n_{v'}}$
17. $\tilde{s}'_u = \tilde{s}_u \oplus \tilde{n_{u'}} \oplus \tilde{n}_v$; $\tilde{s}'_v = \tilde{s}_v \oplus \tilde{n_{v'}} \oplus \tilde{n}_u$

$\mathcal{S}$ returns $\tilde{\tau}_{ae} = [(\widetilde{key}_u, \widetilde{val}_u), (\widetilde{key}_{u'}, \widetilde{val}_{u'}), (\widetilde{key}_v, \widetilde{val}_v), (\widetilde{key}_{v'}, \widetilde{val}_{v'}), (\tilde{g}_u, \tilde{s}'_v), (\tilde{g}_v, \tilde{s}'_u)]$.

Indistinguishability: n_v and $\tilde{n_v}$ are indistinguishable as SE is CPA-secure. Indistinguishability of S_V and $\tilde{S}_V$ follows from the fact that a random string is indistinguishable from the output of G. Similarly, T_V and T_E are indistinguishable from $\tilde{T}_V$ and $\tilde{T}_E$ respectively as both P and F are pseudo-random.

□

8.3 Performance analysis

In this section, we describe the experimental result of our scheme $DCCE$. We have designed a prototype [9] using C++. For both the cloud and the client, we have used a single machine, equipped with Intel Core i7-4770 CPU and with 8-core operating at 3.40GHz. It runs an Ubuntu 16.04 LTS 64-bit operating system with 8GB RAM.

There are three library functions P, F, H together with a symmetric key encryption scheme in our proposed DCCE and we have used *HMAC*, *Salsa20*, *SHA-256*, and *AES* for them respectively from *cryptopp* [31] library.

Datasets: For our experiment, we have taken real-world *SNAP* [55] datasets which is a collection of different types of large networks. We have chosen 'ca-HepPh', 'email-Enron', 'loc-Gowalla', and 'roadNet-CA' each of which is undirected. The number of nodes and edges of them is given in Table 8.1

Table 8.1: Number of nodes and edges

dataset	#Nodes	#Edges	Network type
ca-HepPh	12,008	118,521	Collaboration network
email-Enron	36,692	183,831	Email communication network
loc-Gowalla	196,591	950,327	location based social network
roadNet-CA	1,965,206	2,766,607	Road network

Experimental Results: We have tested our prototype with all four types dataset as in Table 8.1. We can see that the time taken to encrypt the graph data is approximately proportional to $M_e =$

$(|V| + 2|E|)$. Since the main operations to generate the encrypted matrix are the computation of hashes, and random numbers, etc., the result is so. M_e vs encryption time is shown in Figure 8-3. This shows for each M_e the client takes $53.3\mu s$.

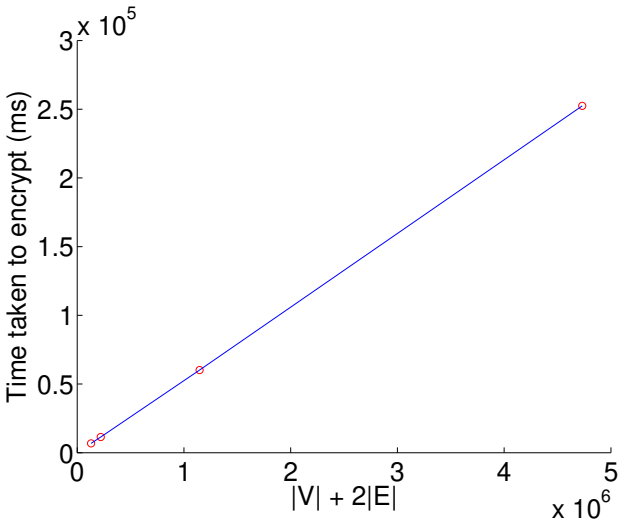


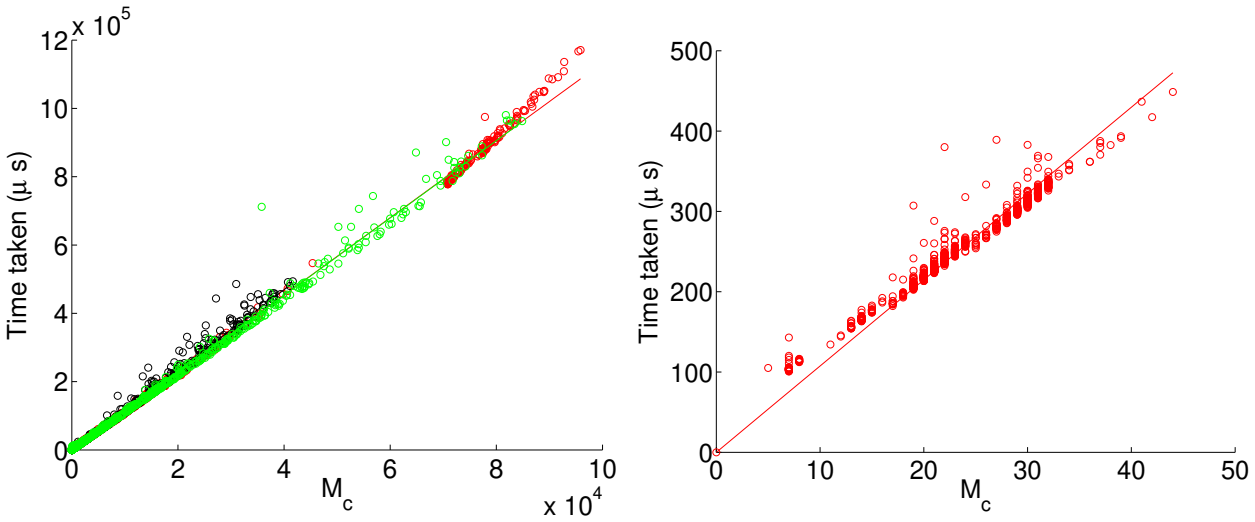
Figure 8-3: Encryption Time on different datasets

For the clustering coefficient query, 1000 random vertices are chosen for each dataset. Since each vertex has a different number of neighbors, query time is different for different vertices. But from experiment, we have seen that query time for a vertex v, is proportional to $M_c = (4|N_v| + \sum_{u \in N_v} |N_u|)$. We have shown query time in Figure 8-4 for all four datasets. *Since roadNet data is very sparse, query time is lesser here*. From the result, it can be seen that the average values of M_cs are $11.33\mu s$, $11.72\mu s$, $11.32\mu s$, and $10.76\mu s$ respectively for 'ca-HepPh', 'email-Enron', 'loc-Gowalla' and 'roadNet-CA'.

Thus, with 196,591 vertices and 950,327 edges, it takes only $1.2s$ whereas with 62 vertices and 159 edges, a CC search requires 18.77s in [101]. This huge improvement is due to use of symmetric key encryption scheme and efficient data structure.

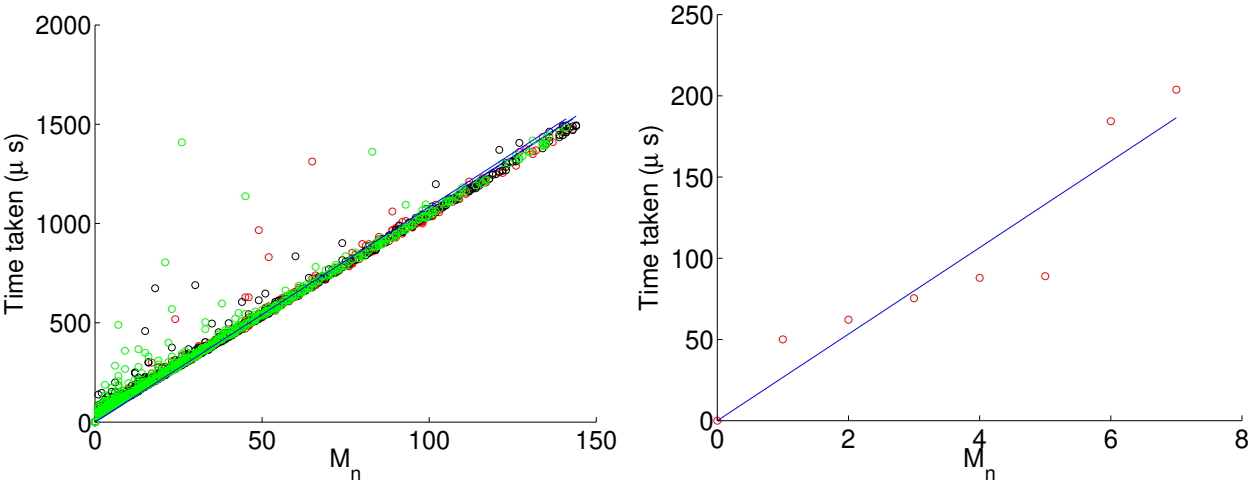
In addition, we have measured time for neighbor query taking random 1000 vertices: Figure 8-5 shows the time taken by the cloud with respect to M_n where $M_n = |N_v|$ for a chosen vertex v. For each dataset, the value of M_n is $10.40\mu s$, $10.46\mu s$, $10.54\mu s$, and $23.68\mu s$ respectively. For roadnet data $M_n = 26.64\mu s$, if we take average before computing line of regression.

Finally, while the addition of a vertex is a concern, the time taken should be and is constant for any vertex. Adding edge is also requires constant time for both the client and the cloud. The time taken to add a vertex and an edge are $69\mu s$ and $124\mu s$ respectively for the client.



(a) HepPh, Enron, Gowalla (b) roadNet-CA

Figure 8-4: CC Query time on different datasets



(a) HepPh, Enron, Gowalla (b) roadNet-CA

Figure 8-5: Neighbor Query time on different datasets

Chapter 9

Conclusion and Future Work

In this chapter of the thesis, we conclude with a summary of the observations found from the works and give a future research direction towards the field of dynamic searchable symmetric encryption.

Dynamic searchable symmetric encryption (DSSE) schemes enable cloud users to search over their encrypted documents uploaded to a cloud server as well as allow them to add new documents or delete some of the existing documents later. In Chapter 4, we have proposed an efficient DSSE scheme Trids which has optimal search time with less deletion leakage compared to other DSSE schemes. We have provided a simulation-based security proof for our scheme Trids. We have implemented a prototype of our scheme Trids and done experiments on real-time datasets to show the efficiency of our DSSE scheme.

In Chapter 5, we have seen that we have successfully presented a publicly verifiable dynamic SSE scheme Srica which is are simple and easy to integrate. Moreover, the VDSSE scheme Srica achieves forward secrecy. In the scheme, we have achieved our target to make efficient for low-resource owner. Due to low computational and communication cost of the owner, we do need an auditor. The presence of the auditor, who verifies the search result, reduces workload of the owner. Our proposed scheme Srica is only for single keyword search queries. There are many other complex queries too. As a future work, one can design complex queried verifiable DSSE scheme. On the other hand, while designing, keeping them forward secret is also a challenging direction of research.

In Chapter 6, we first designed efficient authentication tree DIA tree. Then, we have proposed conjunctive DSE scheme Blasu that is verifiable too. The scheme Blasu uses a forward private single keyword DSE scheme as base. Moreover, our scheme does not use any extra client-storage for verifiability. We have used our designed DIA tree for verifiability. Later, we have shown that the scheme is practical comparing with existing schemes. We can see that most of the verifiable conjunctive search schemes, including ours, use public key encryption for verifiability though some single keyword verifiable schemes use symmetric one. Solving the same problem with with symmetric key encryption technique is still a good open problem in this direction.

In Chapter 7, we have introduced the secure link prediction problem and discussed its security. We have presented three constructions of SLP. The first proposed scheme `SLP-I` has the least

computational time with maximum leakage to the proxy. The second one SLP-II reduces the leakage by randomizing scores. However, it suffers high communication cost from proxy to the client. The third scheme SLP-III has minimum leakage to the proxy. Though the garbled circuit helps to reduce leakage, it increases the communication and computational cost of the cloud and the proxy servers.

Performance analysis shows that they are practical. We have implemented prototypes of first two schemes and measured the performance by doing experiment with different real-life datasets. We also estimated the cost for SLP-III. In the future, we want to make a library that support multiple queries including neighbor query, edge query, degree query, link prediction query etc.

It is to be noted that the cost of computation without privacy and security is far better. The performance has been degraded since we have added security. The performance comes at the cost of security. Throughout this chapter, we have considered unweighted graph. As a future work the schemes can be extended to weighted graphs. Moreover, we have initiated the secure link prediction problem and considered only common neighbors as score metric. As a future work, we will consider the other distance metrics like Jaccard's coefficient, Adamic/Adar, preferential attachment, Katz$_\beta$ etc. and compare the efficiency of each.

In Chapter 8, we have designed a graph encryption scheme Gopas that allows performing clustering coefficient queries over encrypted data. Moreover, our solution Gopas allows performing basic queries including vertex query, edge query, and neighbor query. The solution also supports the addition of a new vertex or edges in the encrypted graph. The results of our prototype, run on real-life data, shows it is efficient and practical.

However, our proposed $DCCE$ scheme Gopas does not support the deletion of a vertex or edges which makes it an append-only scheme. As future work, it is desirable to make such a scheme with both addition and deletion support without compromising the efficiency. Another, interesting direction of research is to enable more types of queries, for example, finding shortest paths, betweenness, etc., on a single encrypted graph. Then, in the future, we can get more general frameworks that allow multiple complex queries or perform other graph queries. In presence of malicious cloud server, developing verifiable versions of the graph problem is also a challenging problem.

Besides the efficiency, it is necessary to discuss scalability when the queries become complex. Though multiple schemes on single keyword search and even multi-keyword search on text data are scalable to very large datasets, when it comes to complex queries, most of the schemes are limited to small datasets only. In complex queries, like fuzzy keyword search, similarity search, substring search, multi-keyword ranked search, etc., public-key encryption schemes are used. This

forces them to have higher execution time and unfit for large databases. Even when the queries become complex or multiple queries are required for a single user, it is desirable to make them scalable for practical large databases. So, designing similar schemes in symmetric-key settings is necessary and very challenging.

Finally, in parallel to the software-based solutions, recent special purpose hardware-based trusted execution environments are helping us to achieve effective and efficient execution of the protocols. Intel Software Guard Extensions (SGX), Intel Trusted Execution Technology (TXT), Trusted Platform Module (TPM), Trusted Computing Base (TCB), ARM TrustZone, etc. are some such hardware. They not only execute with enhanced security but also provide confidentiality and integrity of the applications. Thus designing schemes with them can improve the performance significantly. Though there are few works, like [8], [106], [94], etc., considering such special hardware environments, they are mainly over single keyword search schemes on collection on documents. So, exploring a software and hardware combined approach to address the challenging secure search problem, especially when the queries are complex, is still a new, interesting and necessary direction of research.

Bibliography

[1] Crypto++ benchmark. https://www.cryptopp.com/benchmarks.html.

[2] Daniel Adkins, Archita Agarwal, Seny Kamara, and Tarik Moataz. Encrypted blockchain databases. *IACR Cryptol. ePrint Arch.*, 2020:827, 2020.

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548, 2013.

[4] Imelda Atastina, B Sitohang, G A P Saptawati, and Veronica Moertini. A review of big graph mining research. 180:012065, 03 2017.

[5] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings*, pages 295–308, 2009.

[6] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, and Refik Molva. Publicly verifiable conjunctive keyword search in outsourced databases. In *2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015*, pages 619–627, 2015.

[7] Lars Backstrom, Cynthia Dwork, and Jon M. Kleinberg. Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. *Commun. ACM*, 54(12):133–141, 2011.

[8] Arnab Bag, Sikhar Patranabis, L. Tribhuvan, and Debdeep Mukhopadhyay. Hardware acceleration of searchable encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2201–2203. ACM, 2018.

[9] Gaurav Bansal and Laltu Sardar. Prototypes of the graph encryption scheme supporting clustering coefficient query. *Dropbox repository, https://www.dropbox.com/sh/pzyakffcq75zlxb/AAAW5jGtl23HlF384Qz-BWBxa?dl=0*, 2020.

[10] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, page 9 chapter 4, 2005.

[11] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 506–522, 2004.

[12] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 325–341, 2005.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[14] Christoph Bösch, Andreas Peter, Bram Leenders, Hoon Wei Lim, Qiang Tang, Huaxiong Wang, Pieter H. Hartel, and Willem Jonker. Distributed searchable symmetric encryption. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014*, pages 330–337, 2014.

[15] Angèle Bossuat, Raphael Bost, Pierre-Alain Fouque, Brice Minaud, and Michael Reichle. SSE and SSD: page-efficient searchable symmetric encryption. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 157–184. Springer, 2021.

[16] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1143–1154, 2016.

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

[18] 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 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1465–1482, 2017.

[19] Ning Cao, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 393–402. IEEE Computer Society, 2011.

[20] 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, Denver, CO, USA, October 12-16, 2015*, pages 668–679. ACM, 2015.

[21] 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*, 2014.

[22] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373, 2013.

[23] CERT. 2012 cybersecurity watch survey: How bad is the insider threat? 2012.

[24] Qi Chai and Guang Gong. Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*, pages 917–922, 2012.

[25] 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 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1038–1055, 2018.

[26] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, pages 442–455, 2005.

[27] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 577–594, 2010.

[28] Rong Cheng, Jingbo Yan, Chaowen Guan, Fangguo Zhang, and Kui Ren. Verifiable searchable symmetric encryption from indistinguishability obfuscation. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 621–626, 2015.

[29] 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 *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*, pages 188–207, 2003.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[31] Crypto++:. A free C++ class library of cryptographic schemes. https://www.cryptopp.com.

[32] Reza Curtmola, Juan A. 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 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 79–88, 2006.

[33] Enron. Enron Email Dataset. https://www.cs.cmu.edu/~enron/.

[34] Zhe Fan, Yun Peng, Byron Choi, Jianliang Xu, and Sourav S. Bhowmick. Towards efficient authenticated subgraph query service in outsourced graph databases. *IEEE Trans. Serv. Comput.*, 7(4):696–713, 2014.

[35] Sebastian Gajek. Dynamic symmetric searchable encryption from constrained functional encryption. In *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, pages 75–89, 2016.

[36] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 563–592, 2016.

[37] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT, Madrid, Spain, July 24-26, 2017.*, pages 200–211, 2017.

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

[39] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001.

[40] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 464–479. IEEE Computer Society, 1984.

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

[42] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 310–320, 2014.

[43] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[44] Shunrong Jiang, Xiaoyan Zhu, Linke Guo, and Jianqing Liu. Publicly verifiable boolean query over outsourced encrypted data. In *2015 IEEE Global Communications Conference, GLOBECOM 2015, San Diego, CA, USA, December 6-10, 2015*, pages 1–6, 2015.

[45] Shunrong Jiang, Xiaoyan Zhu, Linke Guo, and Jianqing Liu. Publicly verifiable boolean query over outsourced encrypted data. *IEEE Trans. Cloud Computing*, 7(3):799–813, 2019.

[46] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, pages 94–124, 2017.

[47] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 258–274, 2013.

[48] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 965–976, 2012.

[49] Andreas Kasten, Ansgar Scherp, Frederik Armknecht, and Matthias Krause. Towards search on encrypted graph data. In *Proceedings of the Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn2013) co-located with the 12th International Semantic Web Conference (ISWC 2013), Sydney, Australia, October 22, 2013*, volume 1121 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[50] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[51] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *2009, Kanazawa, Japan, December 12-14, 2009. Proceedings*, pages 1–20, 2009.

[52] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pages 486–498, 2008.

[53] Alan Kuhnle, Nam P. Nguyen, Thang N. Dinh, and My T. Thai. Vulnerability of clustering under node failure in complex networks. *Social Netw. Analys. Mining*, 7(1):8:1–8:15, 2017.

[54] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Yuhong Liu, and Dongxi Liu. Graphse^2: An encrypted graph database for privacy-preserving social search. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 41–54. ACM, 2019.

[55] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[56] Rui Li and Alex X. Liu. Adaptively secure conjunctive query processing over encrypted data for cloud computing. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 697–708. IEEE Computer Society, 2017.

[57] Yuxi Li, Fucai Zhou, Yuhai Qin, Muqing Lin, and Zifeng Xu. Integrity-verifiable conjunctive keyword searchable encryption in cloud storage. *Int. J. Inf. Sec.*, 17(5):549–568, 2018.

[58] David Liben-Nowell and Jon M. Kleinberg. The link prediction problem for social networks. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003*, pages 556–559. ACM, 2003.

[59] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[60] Chang Liu, Liehuang Zhu, and Jinjun Chen. Efficient searchable symmetric encryption for storing multiple source dynamic social data on cloud. *J. Network and Computer Applications*, 86:3–14, 2017.

[61] Pengliang Liu, Jianfeng Wang, Hua Ma, and Haixin Nie. Efficient verifiable public key encryption with keyword search based on KP-ABE. In *Ninth International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2014, Guangdong, China, November 8-10, 2014*, pages 584–589, 2014.

[62] Xueqiao Liu, Guomin Yang, Yi Mu, and Robert H. Deng. Multi-user verifiable searchable symmetric encryption for cloud storage. *IEEE Trans. Dependable Secur. Comput.*, 17(6):1322–1332, 2020.

[63] Zheli Liu, Tong Li, Ping Li, Chunfu Jia, and Jin Li. Verifiable searchable encryption with aggregate keys for data sharing system. *Future Generation Comp. Syst.*, 78:778–788, 2018.

[64] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[65] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. GRECS: graph encryption for approximate shortest distance queries. In *Proceedings of the 22nd ACM SIGSAC*

Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015, pages 504–517, 2015.

[66] Meixia Miao, Jianfeng Wang, Sheng Wen, and Jianfeng Ma. Publicly verifiable database scheme with efficient keyword search. *Inf. Sci.*, 475:18–28, 2019.

[67] Yinbin Miao, Jianfeng Ma, Ximeng Liu, Qi Jiang, Junwei Zhang, Limin Shen, and Zhiquan Liu. VCKSM: verifiable conjunctive keyword search over mobile e-health cloud in shared multi-owner settings. *Pervasive and Mobile Computing*, 40:205–219, 2017.

[68] Yinbin Miao, Jianfeng Ma, Ximeng Liu, Junwei Zhang, and Zhiquan Liu. VKSE-MO: verifiable keyword search over encrypted data in multi-owner settings. *SCIENCE CHINA Information Sciences*, 60(12):122105:1–122105:15, 2017.

[69] Yinbin Miao, Jianfeng Ma, Fushan Wei, Zhiquan Liu, Xu An Wang, and Cunbo Lu. VCSE: verifiable conjunctive keywords search over encrypted data without secure-channel. *Peer-to-Peer Networking and Applications*, 10(4):995–1007, 2017.

[70] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 448–457, 2001.

[71] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 644–655, 2015.

[72] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 639–654, 2014.

[73] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 377–394, 2015.

[74] Wakaha Ogata and Kaoru Kurosawa. Efficient no-dictionary verifiable searchable symmetric encryption. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, pages 498–516, 2017.

[75] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[76] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Cryptographic accumulators for authenticated hash tables. Cryptology ePrint Archive, Report 2009/625, 2009.

[77] PBC LIBRARY , the Pairing-based Cryptography Library. https://crypto.stanford.edu/pbc/".

[78] Ailar Rahimli. Factors influencing organization adoption decision on cloud computing. *International Journal of Cloud Computing and Services Science*, 2(2):141–147, 2013.

[79] Panagiotis Rizomiliotis and Stefanos Gritzalis. ORAM based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop, CCSW 2015, Denver, Colorado, USA, October 16, 2015*, pages 65–76, 2015.

[80] Laltu Sardar. Directory of slp prototype. https://www.dropbox.com/sh/y2obrkefvbrqt05/AAA-nzr1tmK8uJPfVWtXxJFba?dl=0.

[81] Laltu Sardar, Gaurav Bansal, Sushmita Ruj, and Kouichi Sakurai. Securely computing clustering coefficient for outsourced dynamic encrypted graph data. In *13th International Conference on COMmunication Systems & NETworkS, COMSNETS 2021, Bangalore, India, January 5-9, 2021*, pages 465–473. IEEE, 2021.

[82] Laltu Sardar and Sushmita Ruj. Fspvdsse: A forward secure publicly verifiable dynamic sse scheme. In *Provable Security - 13th International Conference, ProvSec 2019, Cairns, QLD, Australia, October 1-4, 2019, Proceedings*, pages 355–371, 2019.

[83] Laltu Sardar and Sushmita Ruj. The secure link prediction problem. *Advances in Mathematics of Communications*, 13(4):733–757, 2019.

[84] Meng Shen, Baoli Ma, Liehuang Zhu, Rashid Mijumbi, Xiaojiang Du, and Jiankun Hu. Cloud-based approximate constrained shortest distance queries over encrypted graphs with privacy protection. *IEEE Trans. Information Forensics and Security*, 13(4):940–953, 2018.

[85] Azam Soleimanian and Shahram Khazaei. Publicly verifiable searchable symmetric encryption based on efficient cryptographic components. *Des. Codes Cryptography*, 87(1):123–147, 2019.

[86] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 44–55, 2000.

[87] 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*, 2014.

[88] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

[89] Shifeng Sun, Joseph K. Liu, Amin Sakzad, Ron Steinfeld, and Tsz Hon Yuen. An efficient non-interactive multi-client searchable encryption with support for boolean queries. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 154–172, 2016.

[90] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 763–780, 2018.

[91] Wenhai Sun, Xuefeng Liu, Wenjing Lou, Y. Thomas Hou, and Hui Li. Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data. In *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, pages 2110–2118, 2015.

[92] Anselme Tueno and Florian Kerschbaum. Efficient secure computation of order-preserving encryption. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 193–207. ACM, 2020.

[93] Peter van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter H. Hartel, and Willem Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management, 7th VLDB Workshop, SDM 2010, Singapore, September 17, 2010. Proceedings*, pages 87–100, 2010.

[94] Viet Vo, Shangqi Lai, Xingliang Yuan, Shifeng Sun, Surya Nepal, and Joseph K. Liu. Accelerating forward and backward private searchable encryption using trusted execution. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part II*, volume 12147 of *Lecture Notes in Computer Science*, pages 83–103. Springer, 2020.

[95] Zhiguo Wan and Robert H. Deng. Vpsearch: Achieving verifiability for privacy-preserving multi-keyword search over encrypted cloud data. *IEEE Trans. Dependable Secur. Comput.*, 15(6):1083–1095, 2018.

[96] Jianfeng Wang, Xiaofeng Chen, Shifeng Sun, Joseph K. Liu, Man Ho Au, and Zhi-Hui Zhan. Towards efficient verifiable conjunctive keyword search for large encrypted database. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, pages 83–100, 2018.

[97] Qian Wang, Kui Ren, Minxin Du, Qi Li, and Aziz Mohaisen. Secgdb: Graph encryption for exact shortest distance queries with efficient updates. In *Financial Cryptography and Data Security - FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, pages 79–97, 2017.

[98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440 EP –, Jun 1998.

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

[100] Zhihua Xia, Xinhui Wang, Xingming Sun, and Qian Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Trans. Parallel Distrib. Syst.*, 27(2):340–352, 2016.

[101] Pengtao Xie and Eric P. Xing. Cryptgraph: Privacy preserving graph analytics on encrypted graph. *CoRR*, abs/1409.5021, 2014.

[102] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 141–158. ACM, 2019.

[103] Lei Xu, Chungen Xu, Joseph K. Liu, Cong Zuo, and Peng Zhang. Building a dynamic searchable encrypted medical database for multi-client. *Inf. Sci.*, 527:394–405, 2020.

[104] Zifeng Xu, Fucai Zhou, Jin Li, Yuxi Li, and Qiang Wang. Graph encryption for all-path queries. *Concurr. Comput. Pract. Exp.*, 32(16), 2020.

[105] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164, 1982.

[106] Attila Altay Yavuz and Jorge Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, volume 9566 of *Lecture Notes in Computer Science*, pages 241–259. Springer, 2015.

[107] Kazuki Yoneyama and Shogo Kimura. Verifiable and forward secure dynamic searchable symmetric encryption with storage efficiency. In *Information and Communications Security - 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings*, pages 489–501, 2017.

[108] Can Zhang, Liehuang Zhu, Chang Xu, Kashif Sharif, Chuan Zhang, and Ximeng Liu. PGAS: privacy-preserving graph encryption for accurate constrained shortest distance queries. *Inf. Sci.*, 506:325–345, 2020.

[109] Rui Zhang, Rui Xue, Ting Yu, and Ling Liu. PVSAE: A public verifiable searchable encryption service framework for outsourced encrypted data. In *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 428–435, 2016.

[110] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 707–720, 2016.

[111] Yao Zheng, Bing Wang, Wenjing Lou, and Yiwei Thomas Hou. Privacy-preserving link prediction in decentralized online social networks. In *Computer Security - ESORICS 2015 - Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 61–80, 2015.

[112] Xiaoyu Zhu, Qin Liu, and Guojun Wang. A novel verifiable and dynamic fuzzy keyword search scheme over encrypted data in cloud computing. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 845–851, 2016.

[113] Cong Zuo, Shifeng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, pages 228–246, 2018.

[114] Cong Zuo, Shifeng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption with forward and stronger backward privacy. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2019.