RUSSIAN-ARMENIAN (SLAVONIC) UNIVERSITY

Mihran M. Hovsepyan

DEVELOPMENT OF SECURE SEARCH ALGORITHMS OVER ENCRYPTED DATA

Thesis

for obtaining a candidate degree in technical sciences in specialty 05.13.05 – "Mathematical modeling, numerical methods and program complexes"

Scientific adviser: Dr. of tech. sc., prof. Gurgen H. Khachatryan

Yerevan 2016

Contents

INTRODUCTION

Actuality of the Subject

In our information society *search* is the main way for accessing the required information. In parallel with Internet development and spreading of *data as a service* (DaaS) [**1**] business model *search* moved from offline to online. We search online as publicly accessible information like learning and research materials, music, photos and videos, products in online stores as well confidential information like personal or commercial documents, financial data, communication history, contacts etc. Actuality of the problems considered in this dissertation are directly connected to the appearance and spread of the DaaS model, that it why it is worth to note how this model has appeared and why it was spread.

Nowadays the efficiency of data management is a fundamental need not only for organizations, but also for individuals and with the incursion of computer devices and new software applications, the amount of digital data produced by an average person or organization in form of emails, multimedia files, records (e.g., financial transactions, healthcare, tax documents, online-shopping, etc.) that they need to store and later access using *search* is dramatically increasing. Effective management of large amount of varied types of data requires powerful tools. And expecting that individuals will be able to develop, set up and administer a complex data management system is not practical and feasible. The same is for small or medium size companies; hiring corresponding professionals for the management of the company's databases is very expensive. DaaS architecture resolves these issues [**1**], [**2**].

The motivation for DaaS architecture was the *software as a service* (SaaS) [**3**] architecture (also referred as *application service provider* (ASP) model). SaaS is a software deployment model in which the software hosted centrally (usually in the web) and licensed on a subscription basis. SaaS has various advantages over the traditional software deployment model; such as the lack of infrastructure configuration and maintenance in the

user side, transparent software and hardware upgrades, lower cost, remote access from anywhere, etc. DaaS is a special case of SaaS when the software provided as a service is a data management and sharing system. In other words DaaS provides to the client various data management and processing functionalities as online services. It overcomes most part of challenges of the traditional database delivery model, mainly related with the setup and maintenance of software and hardware infrastructure.

The main entities of the DaaS model are the following:

- *Data Owner*: The party that allowed doing any kind of operations with the data, since he owns it. It is assumed that the data owner has some storage capabilities and computational power but these are far less than the server's resources.
- *Server*: Stores and administers the owner's data and provides interfaces for remote data management for the owner and other possible users of the data. These interfaces allow a user to do various types of data modification and search queries according to the user's authorization properties (permissions). The maintenance of the data storage and the data administration duties (backups, software and hardware updates, data reorganization, etc.) are entirely taken by the server.
- *Users*: The data owner and other possible parties who have data access permissions. These permissions may be different for different users. For instance one user may be able only to read the data or its part while the other may also has a write access. For instance if the owner of the data is a company its employees and clients may play the role of the users, wherein with different permissions.

The examples of DaaS applications are Amazon S3 storage [**4**], Microsoft Azure Storage [**5**], cloud storages (such as Dropbox [**6**], Google Drive [**7**], Box [**8**], Microsoft OneDrive [**9**], etc.) and this is just a small part of such applications.

As it was told the DaaS architecture provides multiple advantages, though it also has challenges, and one of the main challenges is related with the security of the remotely stored user's data [**10**]. No matter what kind of data it is the most part of the owners view it as a valuable asset and want to keep it secure. To ensure the security of data the server

should provide appropriate guaranties for data confidentiality and integrity. Particularly the service provider has to employ *authentication* (such as Kerberos [**11**]), *authorization & access-control* (such as LDAP [**12**]) and other traditional mechanisms for ensuring the security of their system from the unauthorized and malicious outsiders and from different kind of possible disruptive attacks. These mechanisms are very important, however they do not guaranty the privacy of the data from possible server side adversaries (some malicious program running on the server environment or a curious database administrator), besides that there is always a risk that the data can be stolen due to a bug in the system of the service provider and since in most cases it is not encrypted (Dropbox[1], Yandex Disk, Microsoft OneDrive, Mail.RU Cloud, Google Drive, etc.) the attacker will be able to read it. Serious clients are not satisfied with this level of security and to use these storages they require the providers to involve additional encryption-based security mechanisms to ensure the safety of their data. *Searchable encryption* (SE) schemes are such mechanisms and the **first part of the research** done in the scope of this dissertation (see section 1.1, chapters 2 and 3) refers the issues of those schemes. The *searchable encryption* schemes ensure confidentiality and privacy of the user's data resided in a remote storage without losing possibility of search wherein. There are various publications about these schemes published in recent years [**13**], [**14**], [**15**], [**16**], [**17**], [**18**], [**19**], [**20**], [**21**], [**22**], [**23**], [**24**] etc.

The second part of the research (see section 1.2 and chapter 4) addresses another practical problem related with *search* and *security*. Consider the following: there are two parties, client and server. Client has some pattern p (it can be a piece of text, image or music [**25**]; or some expression which indicates multiple values, such as *regular expressions* [**26**], [**27**]) and the server has data D (can be some text, image or music etc.). The goal of these two parties is one of the following[2]:

1. Check if the data D matches the pattern p (full match).
2. Check if some part of D matches the pattern p (partial match).

[1] Dropbox encrypts stored files only for advanced users. For usual users it keeps files in their original form to be able to optimize storage space by removing duplicate files from different user.

[2] Other variations of the problem are also possible but they are rare and in most cases can be represented as combinations of the listed four.

3. Find the number of parts of D which match the pattern p (count of matches).
4. Find the parts of D which match the pattern p (all matches).

In other words they need to calculate some function $F(p, D)$ which in first two cases has a Boolean value, in the third case the value is a non-negative integer and in fourth case it is a list of positions in D. There is also another requirement, assumed that p and D inputs of the function F are confidential for their owners and none of them wants to share its input with another party. Hence the evaluation of the function F should be done in some oblivious way. This problem is denoted as *secure pattern search* or *secure pattern matching* (later SPM) problem. The problem is a special case of more generic *secure function evaluation* (SFE) problem with two arguments[3] [**28**], [**29**], [**30**], [**31**], [**32**], [**33**], [**34**], [**35**], etc. Though it is considered as a standalone problem in big amount of publications [**36**], [**37**], [**38**], [**39**], [**40**], [**41**], [**42**], [**43**], [**44**], etc.

Constructions intended to solve this problem may found their applications in privacy-preserving genomic computations, remote diagnostics, data mining, face recognition, policy checking, etc. [**31**], [**45**].

Objects of the Research

As was told the first part of the research addresses issues of the schemes which ensure confidentiality and privacy of the user's data deployed into a remote party without losing the possibility of search queries.

One of the important aspects when designing such mechanism is the level of trust, namely how much the data owner trusts the service provider. But even the highest level of trust does not exclude the requirement of such mechanism since no matter how fair is the DaaS provider it can be an object of attacks and hence should guarantee that even being physically stolen the data will be safe.

Making things more secure, which is traditionally done by using encryption, comes with its own costs. Hence another important aspect of these securing mechanisms is the

[3] This problem also denoted as *two-party secure function evaluation* problem.

supported functionality after applying the mechanism. For our case encryption complicates data *search*, *modification* and *sharing* processes. However *search* is the most fundamental functionality when working with data, hence having a securing mechanism which does not support at least some kind of *search* is not practical beforehand. And that is why searchability it the core requirement for these mechanisms, even so different constructions support different types of search queries with different complexities.

The third important challenge for *searchable encryption* schemes is the system restrictions. Usually the restrictions are put on the resources of the system, such as memory, CPU, disk-space, network traffic, etc. So to be practically usable a *searchable encryption* scheme should have reasonable complexities for *search* and *modification* operations and use disk space, memory and network traffic rationally.

Another considerable characteristic of *searchable encryption* schemes is the encryption model it uses: either symmetric or asymmetric. Symmetric SE schemes has better performance characteristics than asymmetric schemes, however they are intended for only one user: the owner of the key, while in asymmetric schemes the data owner who also owns the key can delegate the ability of running queries to third party users providing them the only public key.

So the first object of our research is SE schemes. It also worth to note that the area of SE constructions has been determined by the Defense Advanced Research Projects Agency (DARPA) as a very important one since it can be used to balance the national security and privacy needs in information processing and aggregations systems [**46**]. And in this and many other reasons over the recent years construction and analyze of SE schemes was and remains an important problem in databases and cryptography and there are tons of research materials dedicating this topic [**19**], [**17**], [**18**], [**20**], [**47**], [**48**], [**49**], [**50**], [**51**], [**52**], [**53**] etc.

Besides research there are some applications in the industry (nCryptedCloud [**54**], Sookasa [**55**], boxcryptor [**56**]) intended for securing the data stored in server using encryption, though we found only a single application which supports search functionality

over encrypted data. It is a platform for building secure web applications called “Mylar” [**57**]. There, to search over encrypted user documents the scheme described in “Multi-Key Searchable Encryption” article by R. A. Popa and N. Zeldovich [**58**] is used. That scheme is reviewed in the first chapter of this work.

In the first section of the first chapter a survey of the most practical SE schemes is presented. There we discuss their pros and cons then in the second chapter we present a method for delegating the ability of running search and modification queries to third party users for symmetric SE schemes without proving them the keys of the scheme. To explain the method we designed a new SE scheme by modifying the DSSE “Dynamic Searchable Symmetric Encryption” scheme by S. Kamara and others [**17**]. Finally in the third chapter we review some popular tools intended for securing public cloud storages, after that we introduce an architecture of a system for securing existing public cloud storages using searchable encryption. This architecture was embedded into the new system called “SkyCryptor” [**59**] for making it to support search functionality over encrypted data stored in “Dropbox” and “Google Drive” storages.

The second object of our research is *secure patter matching* problem and the protocols which solve some of its variations. In the second section of the first chapter we briefly review the operation principle of the existing SPM protocols and then in the fourth chapter we introduce a novel SPM protocol, where classical *regular expressions* **Invalid source specified.** are used as patterns and data considered as a text string.

To sum up, the objects of this research are SE schemes, SPM protocols and their practical applications.

Research and Implementation Methods

In the research in general the methods and concepts of classical cryptographic theory and concepts of white-box (later WB) cryptography are used. Besides that for implementing applications for the proposed constructions patterns of object oriented programming (OOP) and C++ as a programming language are used. Also some third party C++ libraries have been used (such as Qt, Boot, CryptoPP, etc.).

Scientific Novelty

1. A novel method for delegating the ability of running search and modification queries to the third party users of a symmetric SE scheme without proving them the keys of the scheme.
2. Architecture which allows to store files in public cloud storages (Google Drive, Dropbox etc.) in encrypted form wherein allows to run secure search queries over those files.
3. A new efficient SPM protocol which allows the client to check if his regular expressions with arbitrary input alphabet matches to the input string of the server.

Practical Significance

The new SE scheme designed and implemented within this dissertation allows the data owner to store his files on the server in encrypted form and give ability to third parties to run secure search and/or modification queries over these encrypted files without providing them the encryption keys. Using the architecture provided in the third chapter this scheme can be used to secure data stored in cloud storages. Such application can be used mostly when working with legally or commercially sensitive files. For example a law firm wants to give ability to his employees to search over the archive of the documents encrypted and stored in the server without providing them the encryption keys.

The designed new SPM protocol due to its high performance and security characteristics may be used in privacy preserving DNA computations.

Practical Implementation

Based on the results of the research *search* functionality was implemented and integrated into the SkyCryptor [**59**] system for giving ability to its users to search for keywords of the encrypted documents stored in Dropbox or Google Drive. The functional allows a user to select a folder in the storage and make it secure by encrypting its files recursively. But before encrypting the files client application automatically invokes keywords of each file and use those to create/update the secure index stored in the SkyCryptor's

server by help of the special modification tokens. This secure index later allows running search queries.

The corresponding implementation certificate is appended to the dissertation.

Provisions Presented to the Defense

- A novel method for delegating the ability of running search and modification queries to the third party users of a symmetric SE scheme without proving them the keys of the scheme.
- A new DSSE [**17**] based symmetric SE scheme which allows delegating the ability of running search and/or modification queries to third parties using the method from the previous point.
- C++ library which implements the new SE scheme and a GUI application which uses this library.
- An architecture which allows to store files in public cloud storages (Dropbox, Google Drive etc.) in encrypted form wherein providing secure search functionality over those files. The architecture was implemented in SkyCryptor system.
- A novel SPM protocol with no asymmetric operations which allows the client to check whether his regular expression Γ with arbitrary input alphabet Σ matches the n-length string $x \in \Sigma^n$.
- A C++ library and an application for demonstrating the designed SPM protocol.

Approbation

The results of the work on dissertation have been presented at:

- International conference - Computer Science and Information Technologies (CSIT) (Yerevan, Armenia) in 2015;
- International workshop – NATO Advanced Research Workshop (ARW) (Aghveran, Armenia), 2015.
- Conference - Science and Technology Convergence (STC) forum (Yerevan, Armenia), 2016.

- Seminars at Russian – Armenian (Slavonic) University, American University of Armenia and at Institute for Informatics and Automation Problems of NAS RA.

Publications

The main topics of this dissertation are published four publications [**60**], [**61**], [**62**], [**63**].

CHAPTER 1. PRIOR ART AND BASE FOR THE CONTRIBUTION

In this chapter we talk about the prior research done on two cryptographic topics which in some manner can be referred as topics related with the *search* and *security*.

The first topic is *searchable encryption* (SE) schemes. In the first section of this chapter we briefly provide the history of SE schemes and review the most practical schemes pointing out their different characteristics (such as dynamicity, multiuserness, efficiency, security, etc.).

The second topic is the two party *secure pattern matching* problem. In the second section of this chapter we briefly discuss characteristics of existing SPM protocols, particularly the supported patterns.

After all in section 1.3 we introduce the approach of using a *WB* implementation of a symmetric encryption algorithm as an alternative to *public-key* from the asymmetric cryptographic model and provide applicability of such approach in the 1-out-of-N oblivious transfer (later OT) protocol introduced in [**64**]. This ability of using WB instead of public key and its application in the OT protocol are used in our constructions and here we provide their brief descriptions.

1.1 SEARCHABLE ENCRYPTION SCHEMES

In opposite to the traditional data processing model (when the data owner deals with the maintenance and administration of the data) in the model when the user data is administered by a remote provider the nature of data processing changes, especially when the trust level in the service provider is very low or perhaps the server is totally untrusted. And in such case the service provider needs to provide appropriate theoretically argued mechanisms of security. Searchable encryption (SE) schemes are such mechanisms [**13**], [**14**], [**16**], [**17**], [**18**], [**19**], [**20**], [**49**], [**50**], [**51**], [**65**], [**66**], [**67**], [**68**] etc. They give ability to store the data in the server in an encrypted form providing search functionality wherein.

Each *searchable encryption* scheme is represented as set of protocols of the following three types:

1. *Initialization protocols*: in this stage based on the scheme's security parameter k and the initial data $\boldsymbol{D}$ (a set of documents) the set of the scheme's keys $\boldsymbol{K}$ and a special encrypted structure $\boldsymbol{C}$ are generated and $\boldsymbol{C}$ is stored to the remote server. This $\boldsymbol{C}$ contains sufficient information to support search and possibly also modification queries to invoke the required information or change it. Though to use that functionality one should have the set of keys $\boldsymbol{K}$ or its some part.
2. *Modification protocols* (file addition, deletion and update): in this stage the user generates corresponding modification token (or tokens) and passes it (them) to the server which updates the encrypted construction $\boldsymbol{C}$ according to the instruction (add, delete, update) hidden in that token(s). In some SE schemes modification tokens may be passed to the server in multiple client-server communication rounds.
3. *Search protocols*: in this stage a search token which usually hides information about one or more keywords is generated by the user and passed to the server. The later one using this token and the encrypted structure $\boldsymbol{C}$ computes the encrypted result of the query and passes it to the user, which later decrypts this results using corresponding keys $\boldsymbol{K}$. In most cases the search query is "find those documents which contain the keyword w" and in that case to search Boolean combinations of multiple keywords (say "find those documents which contain the keyword w_1 and w_2) the user should run multiple queries then merge/union the results, though there are a couple of SE schemes which designed with native support of Boolean queries.

The schemes differ by a bunch of characteristics which refer mainly to one of the following four categories:

1. *Security*: The most important characteristic is the *trust model* for which the scheme intended. The most studied *trust model* is the passive or curious but truthful adversary model. At this model it is considered that the server implements its data and query processing functionality according to the specifications, but there are one or more

malicious individual(s) or spy applications (passive adversaries) on the server-side who has access to the user's data and his search and modification queries (for instance a database administrator or some application running in the server can be considered as a passive adversary). The passive adversary only tries to learn any meaningful information about the stored data without modifying it or corrupting services provided by the server.

Another important characteristic of the security is *forward privacy* (if we search for something (keyword w) and later add a new piece of data (a new document) containing it, the server does not learn that the new document has the keyword we searched before) and backward privacy (query results doesn't contain information about deleted documents).

2. *Multiuserness* is defined based on the fact whether single or multiple users can initially deploy (and later modify) the database and whether single or multiple users can search. Searchable symmetric encryption (SSE) schemes[4] allow only a single user (key holder) to write and read. While in searchable public-key encryption schemes[5] the data owner may provide to third parties corresponding public keys for running search queries.
3. *Dynamicity*: an SE scheme may or may not support data modification queries (add, delete, edit) and based on that it is called static or dynamic. A static SE scheme only in very rare cases can be considered as practical for real-life applications since in most cases the data is dynamic and gets changed frequently that is why the most part of prior research is done on dynamic SE schemes.
4. *Performance*, *memory usage*, *data transfer volume* and client-server *communication rounds*. The most practical SE schemes have search complexity linear at the number of results of the query while the used memory for the secure structure $\boldsymbol{C}$ is proportional to the initial size of data plus total sum of numbers of keywords of each document. Data transfer volume is usually constant and proportional to the size of corresponding

[4] Read about symmetric key cryptography in [**109**].
[5] Read about public key cryptography in [**109**].

query token and the number of client-server communication rounds for the most recent SE schemes is 1, while this numbers increase when Oblivious RAMs are applied on a scheme.

Below we review the most efficient and practical searchable encryption algorithms with different settings and analyze their pros and cons based on above characteristics.

1.1.1 INDEX-BASED SE SCHEMES

Modern searchable encryption schemes are based on the so called *secure* or *encrypted index* approach. In the index-based SE schemes [**14**], [**17**], [**19**], [**20**], [**47**], [**48**], [**50**], [**65**], [**66**] , [**69**], etc. the encryption is done by the data owner(s) (i.e. the client) and then all the encrypted stuff is uploaded into the server.

In these schemes the data $\boldsymbol{D}$ is represented as a set of n files (or documents) $\boldsymbol{f} \stackrel{\text{def}}{=} \{f_1, f_2 \ldots, f_n\}$ and each file has a set of keywords associated with it and a unique identifier (file ID) to differ files between each other. For instance for an office document (word, excel etc.) the set of different words[6] written in those document may play the role of keywords, for e-books the name, authors, chapter names, publisher, etc. can be used and for multimedia files again the name, list of performers, director, year, name of the studio and other fields of meta information can be used as keywords. Choosing correct keywords is important since after encryption only they can be searched. What about file ID then the path of a file relatively to some common for all files root directory or a hash of that path can be used.

Usually we denote by W the set of all keywords of all files and the set the keywords of the i-th file f_i is denoted by $W_i \subseteq W$. The ID of the i-th file will be denoted as fid_i. Also by $|\mathbf{X}|$ we denote the cardinality (number of elements) of the set or dictionary $\mathbf{X}$, by ':=' or '←' signs we denote the assignment operation and by $\langle a_1, a_2, \ldots, a_n \rangle$ we denote the concatenation of $a_1, a_2, \ldots, a_n$.

[6] Actually it is better to use the stem of a word instead of the word itself. And during search operation also first of all change the keyword by its stem. This will give ability to find those documents which contain that keyword in any of its forms (plural, singular, past, perfect, progressive etc.). See more about word stemming in [**110**].

The encrypted structure $\boldsymbol{C}$ in an index-based SE scheme is a pair $(\boldsymbol{c}, \boldsymbol{I})$. Here $\boldsymbol{c}$ is a mapping of file IDs to file ciphertexts, namely $\boldsymbol{c} \stackrel{\text{def}}{=} \{(fid_1, c_1), (fid_2, c_2), \ldots, (fid_n, c_n)\}$, where c_i is the ciphertext of the i-th file encrypted usually by some symmetric encryption algorithm, so the file encryption does not depend on the properties specific to the encryption scheme. And $\boldsymbol{I}$ is the *encrypted* (*secure*) *index* and its construction is the point where the SE schemes differ between each other.

Encrypted index is a data-structure designed in a special way which contains sufficient information about keywords of each file, but to be able to read this information one should have the set of the scheme's encryption keys $\boldsymbol{K}$ or some part of them. As usual data structures these indexes differ by their memory usage, supported operations and their complexities. We will discuss construction of the encrypted indexes and their characteristics in more details during our survey about the existing SE schemes.

While the encrypted index $\boldsymbol{I}$ and ciphertexts $\boldsymbol{c}$ are uploaded into the server the client (data owner) is able to search a keyword and find the list of files with this keyword. Some indexes support search by Boolean expressions of the keywords (like "find list of files with keyword w_2 but not w_1), of course this can be achieved for other indexes as well by running search for each of keywords and performing corresponding operations (intersection, union, difference, etc.) on the result sets, but native support of such type of search may lead to a better performance. Generally the search process works as follows: to search for a keyword w the user generates a search token t_w and sends it to the server, given t_w and using the encrypted index $\boldsymbol{I}$ the server finds the set of file ids $\boldsymbol{fid}_w \stackrel{\text{def}}{=} \{fid_i | w \in W_i\}_{i=1}^{n}$ associated with the keyword w. As a result the search query returns to the user these IDs and probably also corresponding ciphertexts. For some cases the result of a search query may be not the set of file IDs but some encrypted structure which hides those IDs and having corresponding key the user may reveal them out. Hence generally the result of the query is kind of a *data access pattern*, and it is usually called that way.

The modification process (if supported) of the encrypted data and its index $(\boldsymbol{c}, \boldsymbol{I})$ works in a similar way. The user generates corresponding modification token and using it the

server updates both the index $\boldsymbol{I}$ and ciphertexts of the files $\boldsymbol{c}$. These modification tokens contain the ciphertext of the added or edited file and obfuscated information about file ID and its keywords (or only the changed ones).

The two main approaches for building an index are as follows:

- *Forward index*: an index per document (Fig. 1.1(a)) which naturally reduces the search time at least to the number of documents $O(n)$. This is because one index per document has to be processed during a query.
- *Inverted index*: a per keyword index (see Fig. 1.1(b) where $m \stackrel{\text{def}}{=} |W|$ is the number of all keywords). Depending on the amount of information we are OK to leak, the complexity of searching a keyword w can be reduced up to $O(|\boldsymbol{f}_w|)$ in the optimal case, where $\boldsymbol{f}_w$ is the subset of files containing the keyword w.

document id	keywords
1	w_2, w_5, w_7
2	w_1, w_2, w_4, w_6, w_8
...	...
n	w_2, w_5, w_6

(a) Forward index

keyword	document ids
w_1	$2, 3, 9$
w_1	$1, 2, 6, 7, n$
...	...
w_m	$1, 3, 8$

(b) Inverted index

Fig: 1.1

Due to better complexity of search operation most part of modern SE schemes use inverted indexes. Though in both cases indexes use memory proportional to the sum of keywords of all files $O(\sum_{i=1}^{n}|W_i|)$.

1.1.2 SECURITY AND PRIVACY

Before starting the discussion of the specific SE schemes here we talk a little about security and privacy issues. The three statements below seem to be the most intuitive requirements:

- The encrypted database $(\boldsymbol{c}, \boldsymbol{I})$ should leak minimal information about the initial database $\boldsymbol{f}$.

- The search token t_w and the search algorithm itself should leak minimal information to the server about the underlying keyword w.
- The modification token t_{mod} and the modification algorithm should again leak minimal information to the server about the keywords of the added, edited or removed document.

These requirements sound good but there are a couple of important issues. First of all the intuition is not sufficiently accurate. Namely there are lots of important details which affect the security, but they are not mentioned in these statements (e.g., what does the word "leak" and "information" mean here). The details are very important.

Besides formalism there is another issue with this intuition. Here nothing is said about the results of search (data access pattern). Namely, it does not tell whether it is pertinent for a searchable encryption solution to disclose to the server the list of document IDs resulted by searching algorithm. The question has two possible answers. On one hand, there could be an opinion that the disclosure of the data access pattern is fine since server does not know the search query and learning only the names (IDs) of encrypted files which match the search is not a serious problem. And since we want the server to return the identifiers of those encrypted files then it definitely need to know which ones should be returned (note that server doesn't learn the content of the files). On the other hand, there could be another opinion that, theoretically, the data access pattern unfolds some information to the server. Namely, by tracking a sufficient large number of search results some sophisticated statistical attacks can be used by the server to reveal some meaningful information about the client's data and queries. Even more, the argument that the server should know the identifiers of the encrypted documents which match the query for returning them is wrong, because there are various cryptographic OT protocols that allow one side to send and item to another side without knowing the item which is being sent (these are examples of OT and private information reveal protocols [**70**], [**71**], [**64**], [**72**], [**73**], [**74**], [**75**]).

Besides such protocols it is known how to design oblivious RAMs: the system that allows a user to read and write into the memory in a way that the memory device does not know which locations have been accessed. These are examples of oblivious RAMs (ORAM) [**76**], [**77**], [**78**] [**79**]. So ORAMs can be used to hide the data access patter from the server after search is completed. Of course this does not come for free: ORAMs are very slow.

So the answer of the raised question depends on the security and efficiency tradeoff which should be achieved by the scheme. If efficiency is not critical then it is always better to consider a scheme which provides more security and on the other hand if the speed is the priority, then unfolding the data access pattern might be a reasonable price for achieving it.

1.1.3 SYMMETRIC SEARCHABLE ENCRYPTION SCHEMES

The idea of separating encrypted data and its index was not appeared at once. In the paper "Practical Techniques for Searches on Encrypted Data" by Dawn Xiaodong Song, David Wagner and Adrian Perrig from the University of California, Berkeley [**13**] the searchable encryption problem has been explicitly considered for the first time and a scheme with the search time linear at the size of the number of files has been presented.

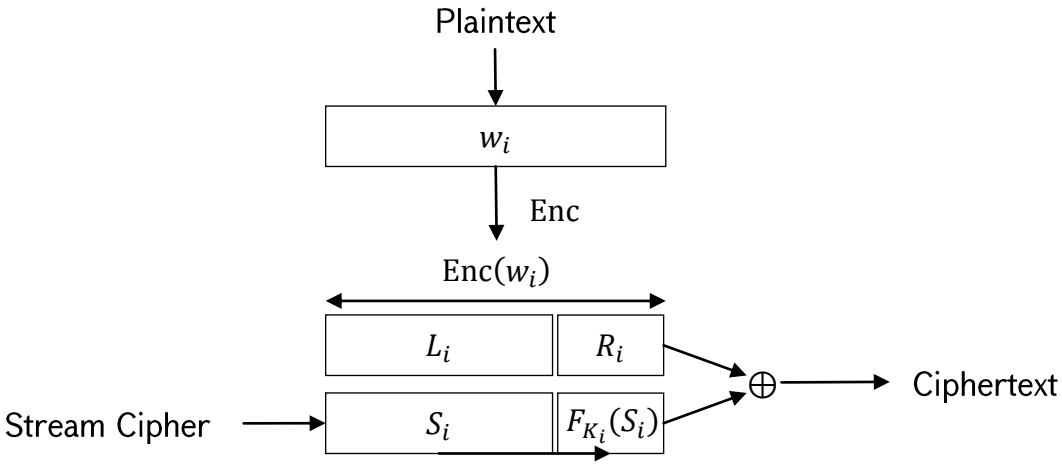

Fig: 1.2

The basic idea was to encrypt the file using any symmetric algorithm word after word in a specific manner see the figure 1.2 above. In this scheme the key owner encrypts each

word w_i of each next file to a fixed size ciphertext $\mathrm{Enc}(w_i)$ and then XOR's it with a pseudo-random stream[7]. To search some word having this construction the user should again encrypt the word using the Enc algorithm, then try to XOR it by each part of the ciphertext checking if the result is equal to the corresponding part of the pseudo-random stream. This construction allows the data owner to give ability to the other party to search a specific word w_i on his behalf by providing only $\mathrm{Enc}(w_i)$ to that party. Besides all this construction supports insertions, deletions and modifications of files in a straightforward way as each file is encrypted word after word. Because of non-practical characteristics such as performance as well as due to the significant information leakage this construction is unusable for practical applications.

Goh [**14**] was the first who proposed an SE scheme with secure-index approach. The proposed secure index allows a user to search that encrypted documents which contain a keyword without decrypting the documents also it allows adding new documents to the encrypted database. In Goh's construction an inverted index based on a Bloom filter [**80**], [**81**] is used which keeps track of each unique keyword. Before being stored into the Bloom filter each of the unique keywords goes through a pseudo-random function (PRF) twice. The purpose of such operation is to make sure that for each of two or more documents, if they associated with the same keyword the code word will represent them differently. The solution requires linear search time, and because of the use of Bloom filter can result in false positives, namely the search may return indexes already deleted from the index. Nevertheless after Goh's work the secure index approach is used as the basis of all upcoming constructions.

Next we review a static[8] symmetric searchable encryption scheme which supports search by Boolean expressions of the keywords. This SSE scheme is suggested in the paper "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries" by David Cash and others in [**66**].

---

[7] Here we do not discuss the details of pseudorandom stream construction, because these details address only issues related to the security of the scheme, and from the point of view our survey are not important.

[8] The construction was later extended to support also database modifications [**82**].

The SSE is a tuple of four algorithms $\mathbf{SSE} \stackrel{\text{def}}{=} (\text{Gen}, \text{EncInd}, \text{Trpdr}, \text{Search})$. Specifications of these functions are given below:

- $\text{Gen}(k)$: In this initialization step the data owner (client) using a random password generates four k-bit[9] uniformly random keys $\boldsymbol{K} = (K_F, K_S, K_X, K_T)$. Also in this step a hash-function with key F (for example HMAC or CMAC) and a symmetric encryption scheme Enc/Dec (say AES) is chosen.
- $\text{EncInd}(\boldsymbol{K}, \boldsymbol{f})$: For the given set of files $\boldsymbol{f}$ client generates a secure index using generated in the first step keys and stores them into the untrusted remote storage (server). Secure index generation steps are the following:
  a. Parse the collection of files $\boldsymbol{f}$ as $V := \left(w_i, \boldsymbol{fid}_{w_i}\right)_{i=1}^{m}$ where $m \stackrel{\text{def}}{=} |W|$ is the count of all unique keywords in all files and $\boldsymbol{fid}_w \stackrel{\text{def}}{=} \{fid_i | w \in W_i\}_{i=1}^{n}$ is the set of file ids which contain the keyword w.
  b. Initialize $XSet$ empty set and A empty dictionary (hash-map).
  c. For every $w \in W$ keyword fill A and $XSet$ as follows.
     a. Initialize an empty list of k-bit strings t;
     b. Set $K_e := F(K_S, w)$, $xtrap := F(K_X, w)$, and $stag := F(K_T, w)$;
     c. For all file IDs $fid \in \boldsymbol{fid}_w$ of the w keyword:
        1. Compute $e := \text{Enc}(K_e, fid)$ and append e to the list t,
        2. Compute $xtag := F(xtrap, fid)$ and add $xtag$ to $XSet$;
     d. Set $A[stag] := t$.
  d. The pair $(A, XSet)$ is the encrypted index and should be stored into the remote server along with the encrypted files[10] $\boldsymbol{c}$.
- $\text{Trpdr}(K_T, K_S, K_x, w_{h_1}, w_{h_2}, \ldots, w_{h_j})$: For searching some Boolean combination of j keywords $w_{h_1}, w_{h_2}, \ldots, w_{h_j}$ the client generates special token (trapdoor) using the secret keys K_S, K_X, K_T in the following way:
  a. Set $stag := F(K_T, w_{h_1})$ and $K_e := F(K_S, w_{h_1})$,

[9] k is a positive integer number (security parameter).
[10] As it was already told in section 1.1.1 "Index-Based SE Schemes" file encryption can be done by any symmetric encryption algorithm.

b. For each $i = 2, \ldots, j$ set $xtrap_i := F(K_X, w_{h_i})$

c. Outputs the $(stag, K_e, xtrap_2, \ldots, xtrap_n)$ as the trapdoor.

- $\text{Search}((stag, K_e, xtrap_2, \ldots, xtrap_n), A, XSet)$: Search is done by the server using the trapdoor generated by the client. The server gets the trapdoor from the client and performs the following actions:

  a. $t := A[stag]$.

  b. For each ciphertext e in t, it computes:

     a. $fid := Dec(K_e, e)$ and then

     b. For all i for 2 to n checks whether fid file contains w_{h_i} by checking if $F(xtrap_i, fid) \in XSet$. And returns fid if it contains only the required keywords.

From the description of the secure index and query (trapdoor) construction and search algorithm it is clear that using this approach any Boolean combination of keywords can be searched after converting it to the disjunctive normal form and creating trapdoors for each conjunction. It is easy to see that both structures of the index $XSet$ and A use $O(N)$ memory, where N is the sum of the keyword numbers of all file ($\sum|W_i|$).

In the real world the data is always dynamic meaning that files can be added or deleted or even the content of each unique file may be changed in the data collection. In this scenario it is desirable to index the data with dynamic methods allowing to efficiently updating the secure index without requiring of re-indexing of the whole database at each update. Below we review one of the most efficient searchable encryption dynamic schemes.

In [**17**] a practical SSE is presented which allows dynamically add and delete files from the secure index in an efficient way. The construction is based on *inverted index* approach. Here the scheme **SSE** consists of 9 algorithms (Gen, Enc, SearchToken, AddToken, DelToken, Search, Add, Del, Dec). In scope of our brief survey here we describe only the index construction and search methods referring the reader to the full paper for details on how to update the secure index.

- $\mathrm{Gen}(k)$: At this stage the data owner selects a symmetric encryption scheme $\mathrm{SYM} \stackrel{\text{def}}{=} (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ then s/he selects uniformly random three k-bit keys K_1, K_2, K_3 and generates the forth key $K_4 \leftarrow \mathrm{SYM}.\mathrm{Gen}(k)$ for encrypting the files. So the scheme has four master keys $\boldsymbol{K} \stackrel{\text{def}}{=} (K_1, \dots, K_4)$, this keys should be never passed to the server, they are used to encrypt files as well as to construct and later modify the secure index. At this stage also three PRFs F, G, P and two keyed random oracles H_1, H_2 are chosen.
- $\mathrm{Enc}(\boldsymbol{K}, \boldsymbol{f})$: Using index generation keys $\boldsymbol{K}$ and the user's document collection $\boldsymbol{f} \stackrel{\text{def}}{=} (f_1 \dots f_n)$ constructs an encrypted index γ and encrypted documents $\boldsymbol{c} \stackrel{\text{def}}{=} (c_1 \dots c_n)$ in a special way:
  a. Let A_s and A_d be search and deletion arrays with enough long length[11] and let T_s and T_d be empty dictionaries. By $\mathbf{0}^l$ is an l-length string of 0's and by $free$ is a word that does not belong to W are denoted.
  b. For each word $w \in W$:
     1. Create the L_w list of nodes $(N_1 \dots N_{|\boldsymbol{f}_w|})$ and chose random location in the array A_s and store these nodes there. Here each node is the following pair:
     $$N_i \stackrel{\text{def}}{=} (H_1(K_w, r_i) \oplus \langle \mathrm{id}_{wi}, \mathrm{ptr}_s(\mathrm{N}_{i+1}) \rangle;\ r_i)$$
     where id_{wi} is the ID of the i-th file in $\boldsymbol{f}_w$, r_i is uniform randomly generated k-bit string, $K_w := P_{K_3}(w)$ and $\mathrm{ptr}_s(N_{|\boldsymbol{f}_w|+1}) \stackrel{\text{def}}{=} \mathbf{0}^{\log|\boldsymbol{A_s}|}$.
     2. Store the pointer to the first node of the L_w list in the search directory by adding the following entry to the dictionary:
     $$T_s[F_{K_1}(w)] := \mathrm{G}_{K_2}(w) \oplus \langle \mathrm{ptr}_s(N_1), \mathrm{ptr}_d(N_1^*) \rangle$$
     where N^* is the dual of N, i.e., the node in A_d whose fourth entry points to N in A_s.
  c. For each file f in $\boldsymbol{f}$:

[11] The length should be longer than the sum of the numbers of all keywords of each file by $z + x$. Where z indicates the number of possible new elements of these arrays and x is the number of cells which will remain unused.

1. Create a list L_f of dual nodes $\left(D_1, \ldots, D_{|W_f|}\right)$ and store them in $|W_f|$ random unused locations in the array A_d (here W_f is the set of the keywords of the file f) defined in the following way: each entry D_i is associated with a word w, and hence with a node N in L_w associated with the same file f. Let N_{+1} be the node following N in L_w and N_{-1} the node previous to N in L_w. Then D_i is defined as follows:

$$D_i \stackrel{\text{def}}{=} \left(H_2\left(K_f, r_i'\right) \oplus \langle \begin{matrix} \text{ptr}_d(D_{i+1}), \text{ptr}_d(N_{-1}^*), \text{ptr}_d(N_{+1}^*), \text{ptr}_s(N), \\ \text{ptr}_s(\text{N}_{-1}), \text{ptr}_s(\text{N}_{+1}), F_{K_1}(w) \end{matrix} \rangle ; \ r_i'\right).$$

Here r_i' is again k-bit uniformly random string, $K_f := P_{K_3}(\text{id}_f)$, where id_f is the identifier of the file f, and $\text{ptr}_d\left(D_{|W_f|+1}\right) \stackrel{\text{def}}{=} \mathbf{0}^{\log|A_d|}$.

2. In the deletion dictionary T_d for f file's ID store L_f list's first node's pointer:

$$T_d\left[F_{K_1}(\text{id}_f)\right] := G_{K_2}(\text{id}_f) \oplus \text{ptr}_s(D_1)$$

d. Choose z unused cells from A_s and A_d from random locations and create a free unencrypted list L_{free}. Let $(F_1, \ldots, F_z)$ and $(F_1', \ldots, F_z')$ be the free nodes in A_s and A_d, respectively. Set

$$T_s[free] := \langle \text{ptr}_s(F_z), \mathbf{0}^{\log|A_s|} \rangle$$

and for $z \geq i \geq 1$ set

$$A_s[\text{ptr}_s(F_i)] := \langle \mathbf{0}^{\log|f|}, \text{ptr}_s(F_{i-1}), \text{ptr}_d(F_i') \rangle$$

where $\text{ptr}_s(F_0) \stackrel{\text{def}}{=} \mathbf{0}^{\log|A_s|}$.

e. Fill the remaining entries of A_s and A_d with random strings.

f. For $1 \leq i \leq n$ calculate $c_i = \text{SYM.Enc}_{K_4}(f_i)$.

g. Return $(\gamma, \boldsymbol{c})$, where $\gamma \stackrel{\text{def}}{=} (A_s, A_d, T_s, T_d)$ and $\boldsymbol{c} = (c_1, \ldots, c_n)$.

- $\text{SrchToken}(w, K_1 \ldots K_3)$: the client uses the first three keys of $\boldsymbol{K}$ and the keyword w and computes the search token:

$$t_s := \left(P_{K_3}(w), G_{K_2}(w), F_{K_1}(w)\right)$$

and returns it.

- $\text{Search}(\gamma, \mathbf{c}, t_s)$: the server takes the encrypted index, the search token and the ciphertexts of documents and returns the search related ciphertexts in the following way
  a. Unfold t_s as (t_1, t_2, t_3) and return an empty list if t_3 is not present in T_s.
  b. Compute
  $$(p_1, p_1') := t_2 \oplus T_s[t_3]$$
  c. Compute $(y_1, r_1) := N_1 := A_s[p_1]$ and decrypt first file ID associated with the keyword hidden in the search token by computing
  $$(\text{id}_1, p_2) := H_1(t_1, r_1) \oplus y_1$$
  d. For each integer $i \geq 2$, decrypt node N_i as above until p_{i+1} becomes equal to $\mathbf{0}^{\log|A_s|}$.

Denote by $I \stackrel{\text{def}}{=} \{\text{id}_1, \text{id}_2, \ldots\}$ be the set of file IDs received in the step [d.] and output them as a result, or find their corresponding ciphertexts $\{c_i\}$ and output them.

Computationally Efficient Dynamic SSE

In [**19**] another efficient and dynamic SE symmetric construction is presented which again provides sublinear search complexity. In this scheme for each document $f \in \boldsymbol{f}$ a separate key K_f is generated for encrypting that document and for each keyword $w \in W$ a key K_w is generated for its encryption. The server stores the searchable representation S_w for the keyword w based on the K_w key and a trapdoor T_w which represents the query from the client for the that keyword. Also lets denote $\boldsymbol{fid}_w \stackrel{\text{def}}{=} \{fid_i \mid w \in W_i\}_{i=1}^{n}$. The construction of the searchable representation for each w keyword is:

$$S_w \stackrel{\text{def}}{=} \left(P_{K_P}(w), M(\boldsymbol{fid}_w), R(w)\right)$$

Where P is a pseudo-random function, and $P_{K_P}(w)$ is used to identify the searchable representation for the keyword w; K_P is a K_w driven key, $M(*)$ is a masking function and $R(w)$ is a function that outputs some information for making possible to unmask $\boldsymbol{fid}_w$. To retrieve the documents associated with the keyword w the client generates the following trapdoor

$$T_w := \left(P_{K_P}(w), R'(w)\right)$$

and sends it to the server. The server first searches $P_{K_P}(w)$ and if it is found, the associated mask $M(\boldsymbol{fid}_w)$ is unmasked using $R(w)$ and $R'(w)$. The server then sends the encrypted data items (files) whose IDs occur in $\boldsymbol{fid}_w$ to the client.

The main idea of the second approach described in [**19**] is to use a chain of the same hash function of length L

$$H^L(\alpha) \stackrel{\text{def}}{=} \underbrace{H\big(H(\dots H(a)\dots)\big)}_{L}$$

by computing the same $H(*)$ function repeatedly L times using α as an argument of the first computation. Let $\boldsymbol{fid}_i(w)$ be the set of file identifiers which contain the keyword w and have been added to the storage during the i-th update. In this cast the searchable representation of the keyword w after updating the storage j times is:

$$S_w \stackrel{\text{def}}{=} \left(P_{K_P}(w), \mathrm{Enc}_{K_1(w)}\big(\boldsymbol{fid}_1(w)\big), H'\big(K_1(w)\big), \dots, \mathrm{Enc}_{K_j(w)}\left(\boldsymbol{fid}_j(w)\right), H'\left(K_j(w)\right)\right)$$

where H' is a hash-function, Enc is an encryption algorithm and $K_i(w) \stackrel{\text{def}}{=} H^{L-ctr}(\langle w, K_w\rangle)$, where ctr is a counter stored in the client-side and incremented each time the storage is updated. As can be seen here each list $\boldsymbol{fid}_j(w)$ is encrypted with a unique key $K_j(w)$ and having it the $K_{j-1}(w), \dots, K_1(w)$ keys of the previously added lists can be computed by the server by computing $j-1$ hash functions, but the next key $K_{j+1}(w)$ cannot be computed since the hash function is not reversible. Based on above explanation assuming that S_w has already been updated j times, the next update of S_w is implemented as follows:

For each unique word $w \in W$:

- Construct $\boldsymbol{fid}_{j+1}(w) := \{ID_j | w \in W_{j,}\}$
- Increment the counter $ctr := ctr + 1$,
- Compute the key $K_{j+1}(w) := H^{L-ctr}(\langle w, K_w\rangle)$,
- Encrypt $\mathrm{Enc}_{K_{j+1}(w)}\left(\boldsymbol{fid}_{j+1}(w)\right)$,
- Send to the server the tuple $\left(P_{k_P}(w), \mathrm{Enc}_{K_{j+1}(w)}\left(\boldsymbol{fid}_{j+1}(w)\right), H'\left(K_{j+1}(w)\right)\right)$ and on the server add that tuple to S_w.

For this construction the search trapdoor for the word w has the following form

$$T_w := \left(P_{k_P}(w), H^{L-ctr}(\langle w, K_w \rangle)\right) := (T_w^1, T_w^2).$$

And the search is implemented as follows: using the trapdoor (T_w^1, T_w^2) and the searchable representation $\boldsymbol{S}$ (hash-map of all S_w where $P_{k_P}(w)$ is the key of the mapping), search if T_w^1 is a key for the hash-map $\boldsymbol{S}$, compute $H'(T_w^2)$, if $H'(T_w^2) = H'\left(K_j(w)\right)$, decrypt $\boldsymbol{fid}_j(w)$ using T_w^2, otherwise keep computing $T_w^2 := H(T_w^2)$ until $H'(T_w^2) = H'\left(K_j(w)\right)$. Repeat the same and compute the identifiers of the previously added documents. Send the merged sets $\boldsymbol{fid}_w \stackrel{\text{def}}{=} \boldsymbol{fid}_i(w) \cup \ldots \cup \boldsymbol{fid}_1(w)$ to the client as a result.

Though this construction has good operation complexities however it has an important limitation: the number of updates is limited by L.

Dynamic Searchable Encryption in Very-Large Databases:

In [**82**] a dynamic searchable encryption scheme is presented which efficiently searches the server-side database of tens of billions rows. A basic algorithm is constructed based on a generic dictionary construction and two other optimized versions are built on top of the basic algorithm. Here we cover the basic construction and refer the interested reader to the original paper for details of the algorithm's next extensions.

Following to the formalization of Curtmola and others [**65**] the database DB is a list of (id_i, W_i) keyword and set where $W_i \subset \{0,1\}^*$ and $id_i \in \{0,1\}^k$, also let W be the union of all W_i and $DB(w)$ be the set $\{id_i : w \in W_i\}$.

A implementation of the DB's main structure: γ dictionary is done by four algorithms $Create$, Get, $Insert$ and $Remove$.

- $Create(\{(l_i, d_i)\})$: takes a list of label and data pairs $\{(l_i, d_i)\}$, where each label is unique, and constructs the data structure γ.
- $Get(\gamma, l)$: returns the data item d with the label l.
- $Insert\big(\gamma, (l, d)\big)$: outputs an updated data structure γ that contains the new pair (l, d).
- $Remove(\gamma, l)$: outputs an updated data structure γ that doesn't contain the label l.

Let F be a variable-input-length PRF, and $\Sigma \stackrel{\text{def}}{=} (Enc, Dec)$ be a symmetric encryption scheme. Below we detail the database setup and search algorithms.

SETUP THE DATABASE

- Chose a key $K \leftarrow \{0,1\}^k$ and allocate the list L.
- For each $w \in W$ keyword generate the keys $K_1 = Enc(K, \langle 1, w \rangle)$ and $K_2 = Enc(K, \langle 2, w \rangle)$.
- Initialize the counter $c = 0$.
- For each $id_i \in DB(w)$ calculate $l := F(K_1, c)$ and $d := Enc(K_2, id_i)$ and increment c by one. After add the pair (l, d) to the list L in lexicographically order.
- Set $\gamma := Create(L)$.
- Outputs the user key K and γ.

SEARCH THE DATABASE

Client: On input (K, w)

- Calculate the keys $K_1 := Enc(K, \langle 1, w \rangle)$ and $K_2 := Enc(K, \langle 2, w \rangle)$ then send them to the server.

Server: Take as an input (K_1, K_2)

- Starting from $c := 0$ and until Get returns NULL set:
  $d := Get(\gamma, F(K_1, c))$ and $id := Enc(K_2, d)$.
- Return the list of revealed ids.

This scheme can be made dynamic with a simple trick of using another dictionary γ^+. This dictionary is firstly empty and after each keyword addition to the database a corresponding (l, d) pairs are being added into it. Searching keyword w is completed by the server first by searching in γ like in the static case and after in γ^+. The labels for γ^+ are computed using a w-based key sent by the client and a counter being incremented at each step. The addition operation for the (id, W) pair requires the client to have the counter's current value for each of the $w \in W$ keywords. To keep these counters an additional dictionary δ is being employed. The directory for each keyword stores the value of the counter (the number of updates where w was involved). δ may be stored either at the client or at the server being every time retrieved by the client for processing update operations. The description of the new (id, W) pair addition algorithm into the database is below:

UPDATE THE DATABASE**:**

- Calculate $K^+ := F(K, 3)$.
- For each $w \in W$ keyword:
  - Compute two keys $K_1^+ := Enc(K^+, \langle 1, w \rangle)$ and $K_2^+ := Enc(K^+, \langle 2, w \rangle)$ as in the previous case.
  - Set $c := Get(\delta, w)$ (if w is not found in δ then set $c := 0$) and calculate the label $l := F(K_1^+, c)$ and data $d := Enc(K_2^+, id)$ then increment c by one.
- After all insert the pair (w, c) into the dictionary δ and insert the pair (l, d) into the γ^+.

The search now implies the searching in both γ^+ and γ dictionaries. The server gets two key pairs (K_1, K_2) and (K_1^+, K_2^+) and performs the following actions.

- For $c = 0$ until Get returns NULL
  Set $d := Get(\gamma, F(K_1, c))$
  And $id := Enc(K_2, d)$
- For $c = 0$ until Get returns NULL
  $d := Get(\gamma^+, F(K_1^+, c))$
  $id := Enc(K_2^+, d)$

This scheme can allow delete operations over the secure index only by keeping the revocation list of all deleted data ids and next check each search result against the revocation list.

1.1.4 PUBLIC-KEY KEYWORD SEARCH

In [**83**] the problem of searching in public-key encrypted data is studied. The main use case of this scheme is considered the email gateway which needs to check emails for specific words, such as «urgent» to route them accordingly. But the mail gateway should not learn anything more about the encrypted messages. So, it is assumed that Bob sends emails to Alice in an encrypted form. For this Bob encrypts the mail he is going to send using the corresponding public-key of Alice after that appends to the ciphertext some footer consisting from encryptions of all keywords of the email. Assuming that the keywords of the email M are $w_1, w_2, \dots w_N$ Bob sends the following:

$$\langle E_{A_{pub}}(M), PKES(K_{pub}, w_1), \dots, PKES(K_{pub}, w_N) \rangle$$

Where K_{pub} is Alice's public-key. The point of this encryption is that Alice can give the email gateway a special trapdoor T_w so the gateway can check whether the encrypted email contains the keyword without learning anything more about the encrypted message or the certain keyword w.

The scheme is defined as a tuple of following algorithms:

- $\mathrm{KeyGen}(s)$: Based on the parameter of the schemes security s generates a pair of public & private keys K_{pub} and K_{priv}.
- $\mathrm{PKES}(K_{pub}, w)$: Encrypts the keyword w using K_{pub} public key in a special way to make it searchable.
- $\mathrm{Trpdr}(K_{priv}, w)$: Using the private key K_{priv} the mail receiver generates a trapdoor T_w for the give key word w.
- $\mathrm{Check}(K_{pub}, S, T_w)$: Using the public key of the mail receiver K_{pub} and the trapdoor T_w for word w checks if $w = w'$ where $S := PKES(K_{pub}, w')$. As a result outputs "Yes" , or "No".

The construction of PEKS scheme is based on a bilinear mapping e from $G_1 \times G_1$ to G_2 where G_1 and G_2 are p prime number's order groups. Namely:

1. Having $g, d \in G_1$, there is an efficient way to compute the value $e(g, d) \in G_2$
2. For any two positive $x, y \in [1, p]$ integers the following equality is true
$$e(g^x, d^y) = e(g, d)^{xy}$$
3. For any d generator of the group G_1 the $e(d, d)$ is a generator for the group G_2.

Two hash functions H_1, H_2 are used as building blocks. The definition of scheme is described as follows:

- $\mathrm{KeyGen}(s)$: Based on the schemes security parameter s the primer number p of the G_1 and G_2 groups is chosen. Also a generator g of the group G_1 and a random string $a \leftarrow Z_p$ are determined in this stage. Finally the pair of private and public keys are defined $K_{pub} := (g, h)$, where $h := g^a$ and $K_{priv} := a$.
- $\mathrm{PKES}(K_{pub}, w)$: For the keyword w first for a random $z \in Z_p$ computes $t := e(H_1(w), h^z),\ t \in G_2$ then outputs $S := \left(g^z, H_2(t)\right)$ as result of encryption.

- $\mathrm{Trpdr}(K_{priv}, w)$: Calculates and returns $H_1(w)^{K_{priv}} \in G_1$.
- $\mathrm{Check}(K_{pub}, S, T_w)$: First unfold S as $(g^z, H_2(t))$ and check if $H_2(e(T_W, g^z)) = H_2(t)$.

The correctness of this scheme comes from the fact that $H_2(e(T_w, g^z)) \stackrel{\text{def}}{=} H_2(e(H_1(w)^a, g^z)) \stackrel{\text{def}}{=} H_2(e(H_1(w), g))^{az}$ and $H_2(t) \stackrel{\text{def}}{=} H_2(e(H_1(w), h^z)) \stackrel{\text{def}}{=} H2eH1w, gaz \stackrel{def}{=} H2eH1w, g\,az$

The security of this SE construction is proven for the chosen keyword attack assuming that given a generator of the group $d \in G_1$ and three values d^x, d^y, d^z of the group G_1 the advantage of an arbitrary polynomial time algorithm is negligible for computing the value $e(d,d)^{xyz} \in G_2$.

1.1.5 MULTI-KEYWORD SEARCHABLE ENCRYPTION SCHEMES

In the multi-key setting [**58**] there are a set of users and each of them has multiple encrypted files and there is a server who is responsible for storing these encrypted files. Each of the users has access to some subset of files. A user can create a new file and give the other users an access to that file by sharing with them the decryption key of that file.

The functionality goal is to allow a user to search keywords over all the files he allowed to access (say n files) even if different keys are used to encrypted these files and different users have encrypted them. Note, the user has all the keys for decrypting all n files, however only one (not n) search query should be sent to the server. The multi-key encryption scheme consists of the following algorithms - (Setup, KeyGen, Delta, Token, Enc, Adjust, Match):

- $\mathrm{Setup}(s)$: Based on the scheme's security parameter s generates the system parameters.
- $\mathrm{KeyGen}(params)$: Takes the system parameters and returns secret keys such as the user's key and keys for file encryption.
- $\mathrm{Delta}(K_1, K_2)$: Calculates the delta between the two input keys.
- $\mathrm{Token}(K, w)$: Generates the search token tk for the specified keyword w using the specified key K.

- $\mathrm{Enc}(K, w)$: Encrypts the word w using the K key and outputs the ciphertext c.
- $\mathrm{Adjust}(tk, \Delta)$: For the search token $tk \stackrel{\text{def}}{=} \mathrm{Token}(k_1, w)$ and delta $\Delta \stackrel{\text{def}}{=} \mathrm{Delta}(K_1, K_2)$ generates a new search token $tk' = \mathrm{Token}(K_2, w)$.
- $\mathrm{Match}(tk, c)$: Checks if a single encrypted word c matches the search query hidden in the token tk and returns a Boolean value.

The key of the user i is denoted as UK_i, and the encryption key of j-th file as K_j. Consider that a user, say Alice (with the key UK_A) has n encrypted files stored on the server, and each of them is encrypted by a separate key K_j where $1 \leq j \leq n$. Alice wants to search the keyword w over all of these n files, so she uses the UK_A key to compute a token for the word w. In order to allow the server to match the token against the files encrypted using the $K_1, K_2, \ldots, K_n$ keys, Alice sends to the server some public information called delta. Alice provides one delta per each of the K_j keys, denoted as $\Delta_{UK_A K_j}$. The server uses $\Delta_{UK_A K_j}$ to convert the search token generated using the UK_A key to a search token for K_j (the process called adjustment). In this way, the server obtains tokens for the word w for all keys $K_1, K_2, \ldots, K_n$ receiving only a single token from Alice. Then it performs a traditional single-key search using these tokens.

As a base for this multi-key SE scheme asymmetric bilinear mappings are used defined as $e: G_1 \times G_2 \to G_T$ between prime order p groups G_1, G_2, G_T. Such mapping has the following properties:

1. Having a pair $(g_1, g_2) \in G_1 \times G_2$, there is an efficient way to compute the value of $e(g_1, g_2)$.
2. For any two positive integers $x, y \in \{1,2, \ldots, p\}$ the following equality is true
$$e\left(g_1^x, g_2^y\right) = e(g_1, g_2)^{xy}$$
3. If g_1 is the generator of the group G_1 and correspondingly g_2 is the generator of the group G_2, then $e(g_1, g_2)$ is the generator of G_T

For hash functions with appropriate output sizes H_1 and H_2 the searchable encryption scheme with multiple keys is defined in the following way:

- $params \leftarrow \text{Setup}(s)$: Based on the security parameter s returns the system parameters: the prime number p, the groups G_1, G_2 and G_T, the linear operator e, and group generators g_1, g_2 and g_T.
- $K \leftarrow \text{KeyGen}(params)$: Returns K key random from the field Z_p.
- $\Delta \leftarrow \text{Delta}(K_1, K_2)$: Returns $\Delta := {g_2}^{K_2/K_1}$; $\Delta \in G_2$.
- $tk \leftarrow \text{Token}(K, w)$: Returns $tk := H_1(w)^K$; $tk \in G_1$.
- $c \leftarrow \text{Enc}(K, w)$: Select a random $r \leftarrow G_T$ and return the pair

$$c := \left(r, H_2(r,\ e(H_1(w), g_2)^K)\right)$$

- $tk' \leftarrow \text{Adjust}(tk, \Delta)$: returns $tk' := e(tk, \Delta) \in G_T$
- $b \leftarrow \text{Match}(tk', c)$: Parses $c := (r, h)$ and checks if h is equal to $H_2(r, tk')$ and outputs 1 or 0 based on the comparison result.

The correctness of this scheme comes from the following observations: having $params \leftarrow \text{Setup}(s)$, $K_1 \leftarrow \text{KeyGen}(params)$, $K_2 \leftarrow \text{KeyGen}(params)$, $\Delta \leftarrow \text{Delta}(K_1, K_2)$, $tk \leftarrow \text{Adjust}\left(\text{Token}\left(K_1, w\right), \Delta\right)$ we can see that tk is equal to $e\left(H_1(w)^{K_1}, {g_2}^{K_2/K_1}\right)$ which is equal to $e(H_1(w), g_2)^{K_2}$, besides that $H_2(r, e(H_1(w), g_2)^{K_2}) = H_2(r, tk)$ which means that $\text{Match}\left(tk, \text{Enc}(K_1, w)\right)$ is equal to $true$.

1.2 SECURE PATTERN MATCHING

In this section another *secure search* problem is considered. While discussing the SE schemes in the previous section we assumed that file keywords or their Boolean expressions are being searched. And generally that kind of search is quite popular, usually to find some materials on the web or in our personal computer we search for the words or phrases about the required material using some search engines [**84**]. But sometimes it may be required to search information (both text and multimedia) using patterns (for example *regular expressions* [**26**] or image and voice patterns [**25**], etc.) so here we consider the extended variant of search, that is search of patterns. Pattern search or pattern matching (the process of detecting patterns in a set of data) is a fundamental operation in data analysis and has various real-life applications in experimental sciences (biogenetics, physics,

psychology, statistics, sociology, etc.), business (market analysis, strategy building, customer behavior analysis, employee productivity analysis etc.) and in any other area the data analysis is required.

More formally pattern search is a two-party operation where one party has a pattern p and the other party has data D. In fact these parties may be the same when one has both the search pattern and data, and this was the setting of the SE problem discussed in the previous section and in chapters 2 and 3. But in this section and in chapter 4 we consider the case when the owner of the data and the owner of the search pattern are different and discuss search security issues which appear in such situation.

The requirement of the pattern matching problem and correspondingly the result of the operation is usually one of the followings:

1. Check if the data D matches the pattern p (full match).
2. Check if some part of D matches the pattern p (partial match).
3. Find the number of parts of D which match the pattern p (count of matches).
4. Find the parts of D which match the pattern p (all matches).

Other variations of the problem also possible but they are rare and in most cases can be represented as combinations of these four. So pattern search is the process of computation of function $F(p, D)$ where the return type of F is either a Boolean value (for first two cases), either an integer (third case) or a set of positions in the data D (fourth case).

As we told pattern searching operation has two inputs[12] and these inputs are hold by two different parties. Depending on the trust level between these parties and confidentiality of each of inputs different security issues may rise during pattern searching operation. For example one of parties or both of them may want to hide their input from the other. Namely neither the owner of the pattern wants to pass the pattern to the owner of the data and nor conversely the data owner wants to pass the data to the pattern owner, but they want to run one of the four pattern matching operations from the above list. In cryptography this problem is denoted as SPM problem and various protocols have been designed in recent

[12] In some cases number of inputs can be more, for example when pattern or the data is split between multiple parties, but these cases are not considered in our work.

years which provide a solution for it with different security characteristics and performance [**42**], [**40**], [**39**], [**43**], [**36**], [**38**], [**41**], [**37**], [**85**], [**86**].

Before this we did not talk about the nature of data and pattern which being searched, because the formulation of the secure pattern search problem does not rely on it. However in all mentioned papers and in other materials we found it is considered that the data D is a string of characters from some finite alphabet Σ ($D \in \Sigma^*$) and the pattern p is a *regular expression* [**26**] or some subclass of *regular expressions* (for instance in paper "5PM Secure Pattern Matching" [**42**] the pattern is a string of characters from Σ which may have at most one wildcard character, i.e. the character which matches any letter[13]). Our secure pattern search protocol described in chapter 4 also operates against these concepts of "data" and "pattern" however we believe that the dramatic rise of amount of remote computations can server as an impetus for research on "secure pattern search" problem for other types of patterns and the semantics of data will also be considered.

Most part of the modern programming languages support of *regular expressions*. A part of the languages (Perl, Ruby, JavaScript, AWK, Tcl etc.) has a built-in support and the others for example C++, Java, .NET languages, Python etc. via standard or 3rd party library. However the expressions supported by these languages are extended and the languages generated by them[14] cannot be denoted by classical *regular expressions*. That is why before going forward it is worth to note that here and later in this thesis by saying *regular expression* we mean its classical definition.

Definition 1.1 (regular expression): *Regular expression* is a string of constant and operator characters (symbols) that denoting string sets and operations between these sets, respectively.

For the finite alphabet Σ

1. $\emptyset$ is a *regular expression* which generations the empty set $L(\emptyset) \stackrel{\text{def}}{=} \emptyset$.

[13] For example if $\Sigma = \{0, 1\}$ and $*$ is the wildcard character, then p can be $0 * 10$ which matches to both 0110 and 0010 strings. The equivalent *regular expression* for this pattern is $0(0|1)10$.

[14] The language generated by a *regular expression* is a set of strings which match that expression.

2. ε is a *regular expression* which generates the set which consist from "empty" string, namely $L(\varepsilon) \stackrel{\text{def}}{=} \{\varepsilon\}$.
3. A letter a from the alphabet Σ is *regular expression* which generates the set $L(a) \stackrel{\text{def}}{=} \{a\}$.
4. For *regular expressions* R_1 and R_2 their concatenation $C \stackrel{\text{def}}{=} (R_1 R_2)$ is a *regular expression* generating the set of all possible concatenations of elements from $L(R_1)$ and $L(R_2)$:

$$L(C) \stackrel{\text{def}}{=} \{xy \mid x \in L(R_1) \text{ and } y \in L(R_2)\}$$

and their alteration $A \stackrel{\text{def}}{=} (R_1|R_2)$ is a *regular expression* generating the union of $L(R_1)$ and $L(R_2)$ sets:

$$L(A) \stackrel{\text{def}}{=} L(R_1) \cup L(R_2)$$

5. If R is a *regular expression* then $K \stackrel{\text{def}}{=} (R^*)$ [Kleene Star [**87**]] generates the following set:

$$L(K) \stackrel{\text{def}}{=} \bigcup_{i=0,1,2\ldots} L(R^i)$$

where $R^0 \stackrel{\text{def}}{=} \varepsilon$, and $R^i \stackrel{\text{def}}{=} \underbrace{(\ldots((RR)R)\ldots R)}_{i \text{ times}}$.

The secure pattern search problem is a special case of more generic two-party secure function evaluation problem. In that problem there is a known for two parties function $F: U \times V \to Z$, the first party has an input argument $u \in U$ and the second party has an input argument $v \in V$ and the objective of these two parties to calculate $F(u, v)$ allowing one or both parties to learn the result, but none of the parties should learn an additional information about the input of the another.

This problem has various real life applications. One such example is searching a number of appearances of a specific DNA pattern among people of some specific group (the FBI wants to allow biogenetical researchers (clients) to find the number of people among the criminals who have a specific (featured by the researchers) pattern in their DNAs

without providing the entire list of DNAs of such people, and without learning which patterns have been searched). In this example the input argument of the first party is the pattern and the input of the second party is the database of DNAs.

The next application is in banking: a person wants to take a credit from a bank. The bank needs to check whether he fits to their requirements, particularly they want to check his credit history storied in Credit Report Agencies (CRA). But full credit report of a person may contain lots of private information as long as the criteria of the bank for giving a credit for the person also may be private. So the bank may want to check his criteria with the CRA without learning the full credit history and without providing the criteria to the CRA. The range of applications of the two-party secure function evaluation problem is not limited to the privacy-preserving genomic computations and credit checking, it can also be used in remote diagnostics, graph algorithms, data mining, medical diagnostics, face recognition, policy checking and in other areas. For more details see [**32**], it gives entire overview of the problem and its applications.

1.3 WHITE-BOX ENCRYPTION AND ITS USAGE IN OBLIVIOUS TRANSFER PROTOCOLS

The aim of traditional (both symmetric and asymmetric) cryptography is to ensure the security of the system in *black-box attack* context. Namely it is assumed that the attacker may have access to the output of cryptographic algorithms but it has no access to algorithm's execution process and environment itself. But in parallel with deploying cryptography in software applications intended for consumer devices (such as mobile phones, personal computers, gaming consoles, etc.) it appeared to be important to take into consideration that an attacker may have access to more information than only the output of crypto algorithms. Currently it should be considered that an attacker has a complete vision of the software execution environment and its implementation. Namely the attacker can analyze the assembler and the process of the software execution (memory chunks, system calls, etc.) using various tools (such as disassemblers, debuggers, simulators, etc.). In the

worst case the adversary may interfere by changing the assembler of execution. Such kind of attack is denoted as white-box attack and the aim of *WB cryptography* (WBC) is implementing an encryption/decryption algorithms (say AES or SAFER+ [**88**]) in a special way to keep the cryptographic assets (keys and other secret information) secure against WB attacks, particularly the cryptographic assets should not be stored in memory of the running program in their original form. To do so these assets should be sewn up into the implementation of the algorithm in such a way that it was impossible to invoke them even having full access to the execution environment.

The figure 1.3 below illustrates high-level concepts of both *black-box* (classical) and *WB* approaches of implementation of a cryptographic algorithm (either encryption or decryption).

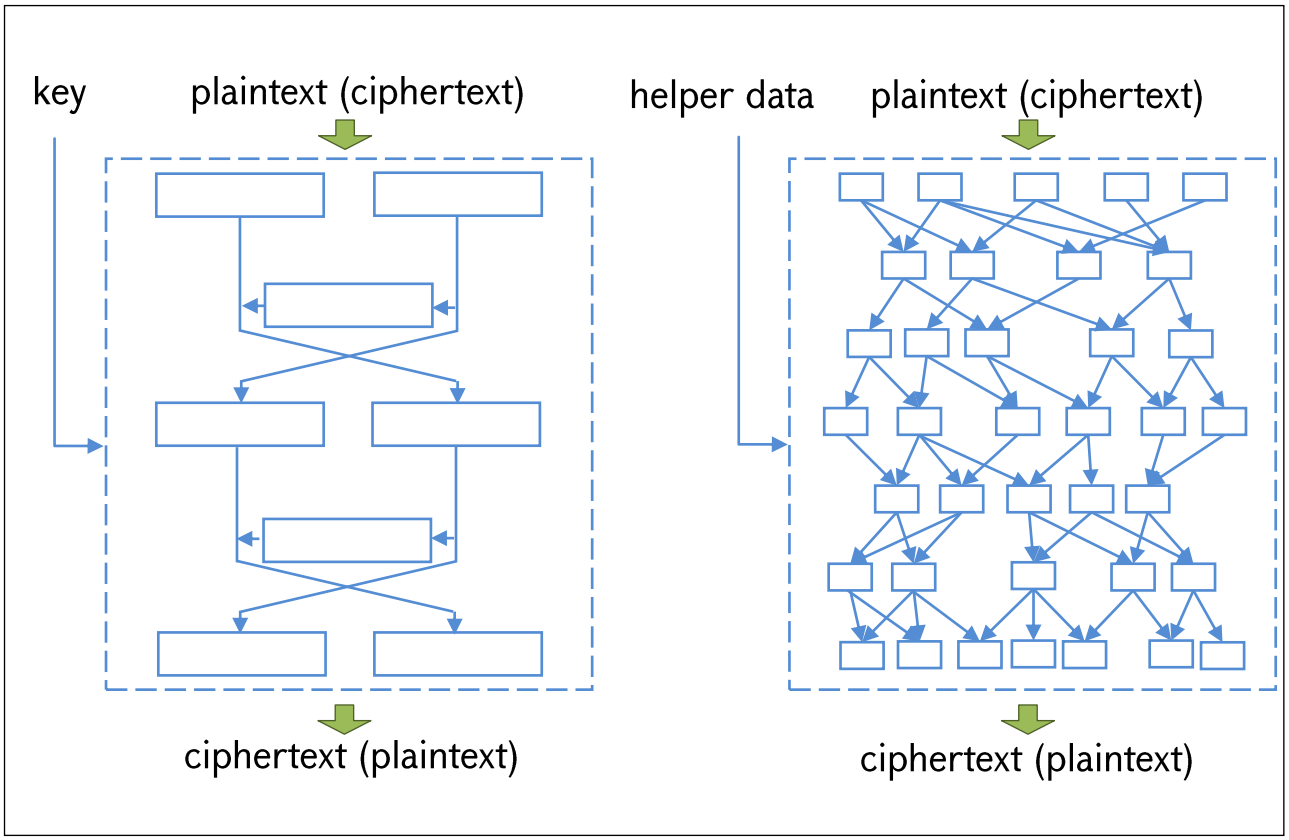

Fig: 1.3

A black-box implementation of a crypto algorithm (the left hand side image) takes as an input both the *key* and the *plaintext (or ciphertext)* and runs the steps of the encryption (decryption) algorithm according to the specifications of the algorithm and returns

ciphertext (or plaintext) as a result while the *WB* implementation of the same algorithm with the same key (the right side image) takes as an input only the *plaintext (or ciphertext)* and some *helper data* and runs some changed encryption (description) algorithm but finally returns the same *ciphertext (or plaintext)* as the black-box implementation was returned. Here the *helper data* is not the *key* but it together with the changed encryption (decryption) algorithm gives the same result as the implementation of the original algorithm with the *key*.

The academic study of WBC was initiated from the seminal work by Chow et al. [**89**] And the major issue with the WB cryptography is the security. Intuitively the security of the implementation of a crypto algorithm against WB attacks (also called as WB encryption/decryption) is measured by the complexity of guessing the secret *key* or making the opposite operation (decryption or encryption) by the WB attacker. And so far all existing AES WB implementations have been broken (are insecure). However the recent work "Design and Cryptanalysis of Secure WB Encryption Based on SAFER+ Algorithm" by G. Khachatryan, M. Kuregyan et al. [**90**] introduces an approach for secure WB implementation of SAFER+ block cipher which is proven that resistant against known WB attacks of AES.

In the proposed approach the WB encryption routine is represented via lookup tables based on the secret keys of encryption. These tables are considered accessible for the attacker in each of the rounds and their main purpose is to hide the original cryptographic keys of the algorithm while performing correct encryption operations. Such WB encryption allows everyone who has access to the WB lookup tables to implement the encryption operation, but since they don't give ability to reveal the original key of encryption hence only the owner of the secret key can decrypt the results of encryption and get the valid plaintext. In this point of view the ***White-Box lookup tables can be considered as a public key because using them everyone can encrypt messages, but only the secret key holder can decrypt them***. This idea plays a fundamental role in our constructions, since it allows changing heavy public-key operations by thousand times faster symmetric key operations.

According to the formalism from [**90**] a WB encryption scheme WBES corresponding to the block cipher algorithm $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ is a list of three $(\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ algorithms performing in the plaintext space $\boldsymbol{M}$, key space $\boldsymbol{K}$ and ciphertext space $\boldsymbol{C}$, such that given a k–bit length master secret key $K \leftarrow_{\mathrm{R}} \boldsymbol{K}$ chosen at random from $\boldsymbol{K}$, the algorithms operate as follows:

- *WB Table Generation*: The owner of the secret key K generates WB encryption tables with respect to K using the table generation public algorithm WBGen:

$$T \leftarrow \mathrm{WBGen}\,(K)$$

- *Encryption*: A random plaintext $M \leftarrow_{\mathrm{R}} \boldsymbol{M}$ chosen from the ciphertext space can be encrypted to the corresponding ciphertext C given the WB tables T and the public encryption algorithm WBEnc:

$$C \leftarrow \mathrm{WBEnc}\,(T; M) \stackrel{\text{def}}{=} \mathrm{WBEnc}_T(P)$$

- *Decryption*: The ciphertext C can be decrypted back to the ciphertext M using the secret key K and the decryption algorithm WBDec:

$$M \leftarrow \mathrm{WBDec}(K; C) \stackrel{\text{def}}{=} \mathrm{WBDec}_K(C)$$

- *Relationship with original algorithms*: For all $M \leftarrow_{\mathrm{R}} \boldsymbol{M}$ and $T \leftarrow \mathrm{WBGen}\,(S)$ the following two statements are correct
  1. $M = \mathrm{Dec}\big(K;\ \mathrm{WBEnc}_T(M)\big) \stackrel{\text{def}}{=} \mathrm{Dec}_K(\mathrm{WBEnc}_T(M))$
  2. $\mathrm{WBEnc}_T(M) = \mathrm{Enc}(K; M) \stackrel{\text{def}}{=} \mathrm{Enc}_K(M)$

In the proposed construction the security parameter $k = 128$, the plaintext and ciphertext spaces and the key space all are comprised of 128-bit length strings.

Definition 1.2: A WB encryption scheme $\mathrm{WBES} \stackrel{\text{def}}{=} (\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ is *secure against key recovery attacks* if given $T \leftarrow \mathrm{WBGen}\,(K)$ it is computationally infeasible to extract the secret key K.

Definition 1.3: A WB encryption scheme $\mathrm{WBES} \stackrel{\text{def}}{=} (\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ is *secure against reverse-engineering attacks* if given $T \leftarrow \mathrm{WBGen}\,(K)$ and any ciphertext $C = \mathrm{Enc}_K(M)$ it is computationally infeasible to compute M.

Definition 1.4: A WB encryption scheme $\mathrm{WBES} \stackrel{\text{def}}{=} (\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ is *computationally secure* if for every $x_0, x_1 \in \{0,1\}^{128}$ the values $\mathrm{WBEnc}_{\mathrm{U}_{128}}(x_0)$ and

$\mathrm{WBEnc}_{\mathrm{U}_{128}}(x_1)$ are computationally indistinguishable, where U_{128} is a uniformly chosen 128-bit length key from the key space.

Definition 1.5: A WB scheme $\mathrm{WBES} \stackrel{\text{def}}{=} (\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ is considered as *secure* if it complies all three security definitions above.

Referring the reader to the paper [**90**] for more details later in our work we will assume that a secure WB encryption scheme $\mathrm{WBES} = (\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ exists.

The next construction block used in constructions designed within this dissertation is *1-out-of-N oblivious transfer* (*OT*) *protocol*. This is a two party (client and server) protocol intended for solving the following problem. The client has a number i from 1 to n and the server has a list of n secrets $(s_1, s_2, \ldots, s_n)$, they should communicate and after that the client should learn the i-th secret s_i, but the server should not learn the i and the client should not learn any information about the other secrets apart from s_i. The figure 1.4 below illustrates how the OT protocol works.

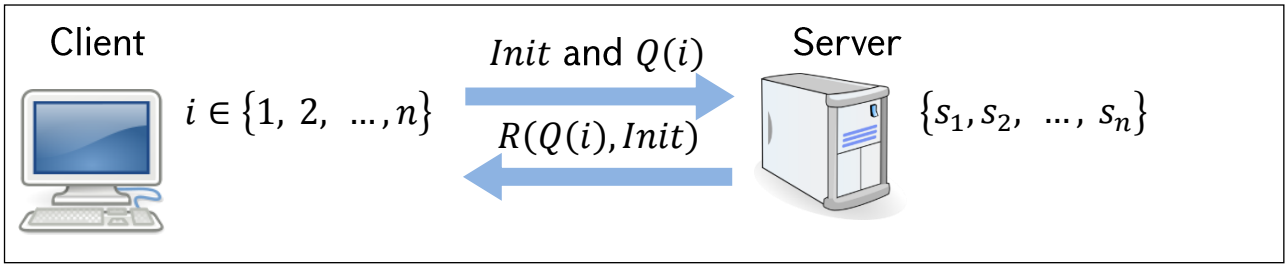

Fig: 1.4

Here:

- $Init$ – Some initialization data.
- $Q(i)$ – Query token for number i which hides i from the server.
- $R(Q(i), Init)$ – Response token which contains enough info to invoke s_i, but nothing else.

OT protocols [**91**], [**92**], [**93**] are important cryptographic primitives having numerous applications and particularly playing an essential role in *secure function evaluation* protocols. On the other hand existing OT protocols are based on computationally expensive public-key operations which remains the main obstacle for employing such protocols in applications where the speed of computations is critical. But the possibility of using lookup tables from WB implementation of SAFER+ encryption algorithm as an alternative to public-

key made it possible to design OT protocols which use no asymmetric crypto operations [**94**]. The new SPM protocol introduced in chapter 4 is based on that OT protocol.

CHAPTER 2. A METHOD FOR DELEGATING SEARCH FUNCTIONALITY TO THIRD PARTIES FOR SSE SCHEMES

In this chapter we introduce a novel method for delegating querying functional of a symmetric SE scheme to the third parties without providing them any of the scheme's master keys. The method makes possible for third-parties to search on the user's encrypted database as well as update the database by adding (and deleting) new documents without compromising the security of existing documents or corrupting the practical aspects of scheme usage. The access to the query related files as well as the permission of updating the database should be handled via access control mechanism set by the user and managed by the server which is considered to be *honest but curious*. Only the master key owner is able to decrypt the third-parties added files. The underlying idea of our approach is to employ WB cryptography techniques to delegate the encryption functionality to third-parties without revealing the secret key.

To demonstrate the method we construct a novel searchable encryption scheme intended for multiple users based on the scheme called Dynamic Symmetric Searchable Encryption [**17**] where the data owner can grant third-parties to search on his encrypted database, as well as update the database without violating the master key security or adding extra communication and/or computation efforts for each search.

2.1 DEFINITIONS AND NOTATIONS

Here we use the same notations as in the first chapter, namely the data is a set of n documents (or files) $\boldsymbol{f} \stackrel{\text{def}}{=} \{f_1, f_2 \ldots, f_n\}$ and each of them has a set of keywords W_i associated with it. By W we denote the set of all keywords from all files and for $w \in W$ by $\boldsymbol{f}_w \subseteq \boldsymbol{f}$ we denote the set of files associated with the keyword w. The ID of the i-th file is denoted by fid_i. Also by $|\boldsymbol{X}|$ we denote the cardinality (number of elements) of the set or dictionary $\boldsymbol{X}$ and for a binary string w by $|w|$ we denote its bit-length. ':=' and '←' signs denote the assignment operation and $\langle a_1, a_2, \ldots, a_n \rangle$ is the concatenation of $a_1, a_2, \ldots, a_n$ strings. The set

of the binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. We write $x \leftarrow \mathcal{X}$ to represent that an element x being sampled from the distribution $\mathcal{X}$, and $x \leftarrow_{\mathrm{R}} \boldsymbol{X}$ to represent an element x being sampled uniformly at random from a set $\boldsymbol{X}$. Given a sequence of elements $\boldsymbol{v}$ we refer to its i^{th} element either as $\boldsymbol{v}[i]$. If the pair (s, v) is in the dictionary T, then $T[s]$ is the value v associated with s and we write $s \in T$ to mean that there exists a pair in T with search key s. And finally we will assume that $\mathrm{WBES} = (\mathrm{WBGen}, \mathrm{WBEnc}, \mathrm{WBDec})$ is a secure WB encryption scheme as it was defined in section 1.3.

The dynamic SE scheme for multiuser users and one owner is a tuple of fourteen polynomial-time algorithms $\mathrm{SSE} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{GenSearchGateway}, \mathrm{GenUpdateGateway}, \mathrm{SearchToken}, \mathrm{GatewaySearchToken}, \mathrm{AddToken}, \mathrm{GatewayAddToken}, \mathrm{DelToken}, \mathrm{GatewayDelToken}, \mathrm{Search}, \mathrm{Add}, \mathrm{Del}, \mathrm{Dec})$ such that:

1. **Gen:** generates the secret keys for constructing the secure index and encrypting the files as well as for generating search and modification tokens. Performer is the data owner.
2. **Enc:** Takes the set of documents and the secret keys and outputs the set of ciphertexts of these documents and secure index of the scheme. Performer is the data owner.
3. **GenSearchGateway:** Using the master keys of the scheme generates a special gateway which can be shared with third-party users and be used for search token generation. The data does not reveal the secret key. Performer is the data owner.
4. **GenUpdateGateway:** Again using the master keys of the scheme generates a special gateway which can be shared with third-party users and for generation of update tokens. Update tokens allow adding a new file or deleting an existing file. The gateway does not reveal the master keys. Performer is the data owner.
5. **SearchToken:** Using the master keys and a keyword generates a special token corresponding to the given keyword. This *search token* can be used by the server to make search over encrypted index. Performer is the data owner.

6. **GatewaySearchToken:** Again generates the search token for a given keyword but this time using the search gateway. Performer is the third-party user who has a gateway for generating search tokens generated by the data owner using the **GenSearchGateway** algorithm.
7. **AddToken:** Using the scheme's master keys for the given file generates a special *addition token* which can be used to update the secure index for adding information about the keywords of the new file there. Performer is the data owner.
8. **GatewayAddToken:** Again generates a token for file addition but this time using the update gateway. Performer is the third-party user who has a gateway for generating update tokens generated by the **GenUpdateGateway** algorithm.
9. **DelToken:** Using the scheme's master keys generates a *deletion token* for the given file identifier which can be used to update the secure index by deleting from it information about the file. Performer is the data owner.
10. **GatewayDelToken:** Again based on the file identifier generates a deletion token but this time using the deletion gateway generated by **GenUpdateGateway** algorithm. Performer is the third-party user who has the update gateway.
11. **Search:** Takes the search token generated either by the **SrchToken** or **GatewaySrchToken** and returns the list of file identifiers (or their ciphertexts) containing the keyword hidden in the token. Performer is the server.
12. **Add:** Takes the add token generated either by the **AddToken** or **GatewayAddToken** and adds to the secure index information about the new file of the token. Performer is the server.
13. **Del:** Takes the search token generated either by the **DelToken** or **GatewayDelToken** and deletes from the secure index information about the file for which the token was created. Performer is the server.
14. **Dec**: Using the passed key decrypts the ciphertext of a single file returned as a result of **Search** and returns the plaintext document. Performer is the data owner or a third party user how has the key for decrypting the files.

2.2 THE ALGORITHM

Our multiuser SE Scheme is an extension of Dynamic Symmetric Searchable Encryption scheme [**17**] which was briefly reviewed in the section 1.1, though the approach of using WB cryptography can be used to extend most of the existing symmetric SE schemes to obtain a multiuser SE scheme. The reason why we have concentrated on this solution is that it seems to be the most efficient and dynamic scheme providing a reasonable tradeoff between efficiency and security.

Let's consider that $\mathrm{SYM} \stackrel{\text{def}}{=} (\mathrm{Gen},\ \mathrm{Enc},\ \mathrm{Dec})$ is a symmetric encryption scheme and

$$F: \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k,$$
$$G: \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k,$$
$$P: \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k,$$
$$R: \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k,$$
$$M: \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k,$$
$$L: \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k,$$

are PRFs, where k is the security parameter of the scheme . Let's consider also that

$$H_1 : \{0,1\}^* \to \{0,1\}^*$$

and

$$H_2 : \{0,1\}^* \to \{0,1\}^*$$

are random oracles. Besides that let

$$WBF: \mathbf{T}_F \times \{0,1\}^* \to \{0,1\}^k$$

be the white-box implementation of the function F with the input parameter $T_F \in \mathbf{T}_F$, which is a WB table. And in the same way

$$WBG: \mathbf{T}_G \times \{0,1\}^* \to \{0,1\}^k$$
$$WBP: \mathbf{T}_P \times \{0,1\}^* \to \{0,1\}^k$$
$$WBR: \mathbf{T}_R \times \{0,1\}^* \to \{0,1\}^k$$
$$WBM: \mathbf{T}_M \times \{0,1\}^* \to \{0,1\}^k$$
$$WBL: \mathbf{T}_L \times \{0,1\}^* \to \{0,1\}^k$$

are the WB analogues of the functions G, P, R, M, L.

The 14 algorithms SSE = (Gen, Enc, GenSearchGateway, GenUpdateGateway, SearchToken, GatewaySearchToken, AddToken, GatewayAddToken, DelToken, GatewayDelToken, Search, Add, Del, Dec) of the multiuser SE scheme are described below:

- $\mathrm{Gen}\,(k)$: sample six k-bit strings $K_1, K_2\,, K_3, K_5\,, K_6, K_7$ uniformly at random and generate $K_4 \leftarrow \mathrm{SYM.Gen}(k)$. Output $\boldsymbol{K} \stackrel{\text{def}}{=} (K_1, K_2\,, K_3, K_4, K_5\,, K_6, K_7)$.

- $\mathrm{Enc}(\boldsymbol{K}, \boldsymbol{f})$: takes the index generation keys $\boldsymbol{K}$ and the collections of user's files $\boldsymbol{f} = (f_1, \dots, f_n)$ and returns an encrypted index γ and the set of encrypted documents $\boldsymbol{c} = (c_1, \dots, c_n)$ in the following way:
  a. Let A_s and A_d be arrays and let T_s and T_d. Let also $\mathbf{0}^l$ be the l-bit string of zeros and $free$ be a word that does not belong to W.
  b. for each keyword $w \in W$:
     1. Create the L_w list of nodes $(N_1 \dots N_{|\boldsymbol{f}_w|})$ then chose random locations for them in A_s and store them there. Each node is the following pair:
     $$N_i \stackrel{\text{def}}{=} (H_1(K_w, r_i) \oplus \langle \mathrm{id}_{wi}, \mathrm{ptr}_s(\mathrm{N}_{i+1})\rangle;\ r_i)$$
     where r_i is a uniformly random generated k-bit string and $K_w := P_{K_3}(w)$ and $\mathrm{ptr}_s(N_{|\boldsymbol{f}_w|+1}) \stackrel{\text{def}}{=} \mathbf{0}^{\log|A_s|}$.
     2. Store the pointer of the first node of the list L_w in the search directory by adding the following entry:
     $$T_s[F_{K_1}(w)] := G_{K_2}(w) \oplus \langle \mathrm{ptr}_s(N_1)\,, \mathrm{ptr}_d(N_1^*)\rangle$$
     where N^* is the dual of N, i.e., the node in A_d which fourth entry points to N in A_s.
  c. for each file f in $\boldsymbol{f}$:
     1. Create a list L_f of dual nodes $\left(D_1, \dots, D_{|W_f|}\right)$ then chose random locations in the A_d and store those nodes there (here W_f is the set of keywords of the file f). A dual node D_i is associated with a keyword w, and hence with a node N in L_w associated with the same file f. Let N_{+1} be the node following

N in L_w and N_{-1} the node previous to N in L_w. Then D_i is defined as follows:

$$D_i \stackrel{\text{def}}{=} \left(H_2(K_f, r_i') \oplus \langle \begin{matrix}\text{ptr}_d(D_{i+1}), \text{ptr}_d(N_{-1}^*), \text{ptr}_d(N_{+1}^*), \text{ptr}_s(N), \\ \text{ptr}_s(\text{N}_{-1}), \text{ptr}_s(\text{N}_{+1}), F_{K_1}(w)\end{matrix} \rangle ; r_i'\right)$$

Here r_i' is a k-bit uniformly random sting. $K_f := L_{K_7}(\text{id}_f)$, and $\text{ptr}_d\left(D_{|W_f|+1}\right) = \mathbf{0}^{\log|A_d|}$.

2. To remember the entry point of the list L_f store it's first element's pointer in the deletion table T_d:

$$T_d[R_{K_5}(\text{id}_f)] := M_{K_6}(\text{id}_f) \oplus \text{ptr}_d(D_1)$$

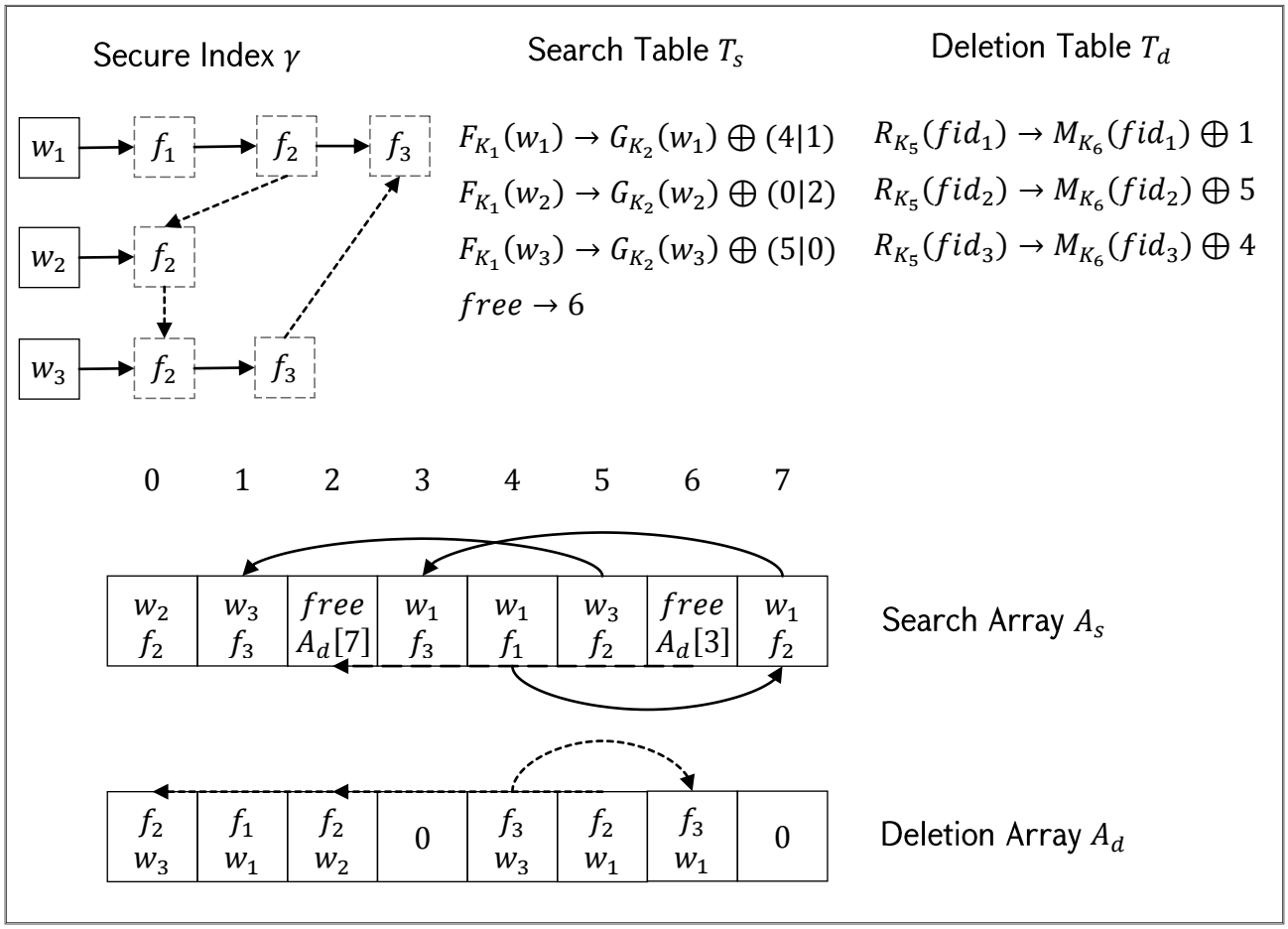

Fig: 2.1

d. Create the list L_{free} for $free$ word by choosing the z unused cells located in random locations in A_s and in A_d. Let $(F_1, \dots, F_z)$ and $(F_1', \dots, F_z')$ be that nodes correspondingly in A_s and A_d. Set

$$T_s[free] := \langle \mathrm{ptr}_s(F_z), \mathbf{0}^{\log|A_s|} \rangle$$

and for $z \geq i \geq 1$ set

$$A_s[\mathrm{ptr}_s(F_i)] := \left(\mathrm{ptr}_s(F_{i-1}); \mathrm{ptr}_d(F_i')\right)$$

where $\mathrm{ptr}_s(F_0) = \mathbf{0}^{\log|A_s|}$

e. Fill the remaining entries of A_s and A_d with random strings.

f. For $1 \leq i \leq n$ set $c_i := \mathrm{SYM.Enc}_{K_4}(f_i)$

g. Output $(\gamma, \mathbf{c})$, where $\gamma \stackrel{\text{def}}{=} (A_s, A_d, T_s, T_d)$ and $\boldsymbol{c} \stackrel{\text{def}}{=} (c_1, \ldots, c_n)$.

In the figure 2.1 above a small example of secure index γ is illustrated.

- $\mathrm{GenSrchGateway}(K_1, K_2, K_3)$: the client takes the key (K_1, K_2, K_3) and returns the search gateway in the following way:
  1. Computes the WB tables corresponding to the PRF $F: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ for the specified key K_1
  $$T_F := \mathrm{WBGen}(F, K_1)$$
  2. Computes the WB tables corresponding to the PRF $G: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ for the specified key K_2
  $$T_G := \mathrm{WBGen}(G, K_2)$$
  3. Computes the WB tables corresponding to the PRF $P: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ for the specified key K_3
  $$T_P := \mathrm{WBGen}(P, K_3)$$

  Outputs the search gateway $\boldsymbol{T}_s \stackrel{\text{def}}{=} (T_F, T_G, T_P)$.

- $\mathrm{GenUpdGateway}(\boldsymbol{K})$: the key owner uses the master keys $\boldsymbol{K}$ and generates the $(T_F, T_G, T_P) := \mathrm{GenSrchGateway}(K_1, K_2, K_3)$ according to the previous algorithm, then generates the forth WB tables corresponding to the $\mathrm{SYM.Enc}$ algorithm and the selected key K_4 as $T_{\mathrm{Enc}} := \mathrm{WBGen}(\mathrm{SYM.Enc}, K_4)$ after which generates the WB tables T_R, T_M, T_L corresponding to the functions R, M, L and keys K_5, K_6, K_7.

  Outputs the update gateway $\boldsymbol{T}_u \stackrel{\text{def}}{=} (T_F, T_G, T_P, T_R, T_M, T_L, T_{\mathrm{Enc}})$.

- $\mathrm{SrchToken}(K_1, K_2, K_3, w)$: the key owner uses the key K_1, K_2 and K_3 and keyword w and outputs the following search token

$$t_s := \left(P_{K_3}(w), G_{K_2}(w), F_{K_1}(w)\right)$$

- $\mathrm{GatewaySrchToken}(\boldsymbol{T}_s, w)$: the third-party user uses the search gateway $\boldsymbol{T}_s \stackrel{\text{def}}{=} (T_F, T_G, T_P)$ and the keyword w and outputs the next search token

$$t_s := \left(WBP_{T_P}(w), WBG_{T_G}(w), WBF_{T_F}(w)\right)$$

as can be seen, two search tokens for the same keyword generated either using $\mathrm{SrchToken}$ or using $\mathrm{GatewaySrchToken}$ algorithm are identical.

- $\mathrm{Search}(\gamma, \boldsymbol{c}, t_s)$: the server uses the encrypted index, the search token and the collection of the ciphertexts of all files and returns the search related file IDs (or their ciphertexts) in the following way
  a. Parses t_s as (t_1, t_2, t_3) and if $t_3 \notin T_s$ returns an empty list of file IDs (or ciphertext) otherwise:
  b. Reveal the pointer to the result list's first node by computing

$$(p_1, p_1') := t_2 \oplus T_s[t_3]$$

  c. Then parse $(y_1, r_1) := N_1 := A_s[p_1]$ and decrypt it with t_1 by computing

$$(\mathrm{id}_1, p_2) := H_1(t_1, r_1) \oplus y_1.$$

  d. For $i \geq 2$, decrypt node N_i as above while $p_i \neq \mathbf{0}^{\log|A_s|}$.
  e. Output the identifiers revealed during the algorithm $\{\mathrm{id}_1, \mathrm{id}_2, \ldots\}$ or the ciphertexts of the corresponding files.

- $\mathrm{AddToken}(\boldsymbol{K}, f)$: computes a special addition token for the specified file f using the master keys $\boldsymbol{K}$ of the scheme.
  1. Let W_f be the set of keywords of the new file.

2. Compute

$$t_a := \left(R_{K_5}(\mathrm{id}_f), M_{K_6}(\mathrm{id}_f), \lambda_1, \ldots, \lambda_{|W_f|}\right)$$

where for all $1 \leq i \leq |W_f|$:

$$\lambda_i := \begin{pmatrix} F_{K_1}(w_i), G_{K_2}(w_i), \langle \mathrm{id}_f, \mathbf{0} \rangle \oplus H_1\left(P_{K_3}(w_i), r_i\right), \mathrm{r_i}, \\ \langle \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, F_{K_1}(w_i) \rangle \oplus \mathrm{H}_2\left(L_{K_7}(\mathrm{id}_f), r_i'\right), \mathrm{r_i'} \end{pmatrix}$$

Where r_i and r_i' are random k-bit strings.

3. Compute $c_f \leftarrow \mathrm{SYM.Enc}_{K_4}(f)$
4. Output (τ_a, c_f)

- $\mathrm{GatewayAddToken}(\boldsymbol{T}_u, f)$: for a new file f uses the $\boldsymbol{T}_u \stackrel{\text{def}}{=} (T_F, T_G, T_P, T_R, T_M, T_L, T_{\mathrm{Enc}})$ update gateway and computes the addition token similar to the one generated by the $\mathrm{AddToken}(\boldsymbol{K}, f)$ in the following way:
  1. Let $W_f \stackrel{\text{def}}{=} \left(w_1, \ldots, w_{|W_f|}\right)$ be the set of keywords of the file f.
  2. Compute $t_a := \left(WBF_{T_R}(\mathrm{id}_f), WBG_{T_M}(\mathrm{id}_f), \lambda_1, \ldots, \lambda_{|W_f|}\right)$, where for all $i \in \{1 \ldots |W_f|\}$:

$$\lambda_i := \begin{pmatrix} WBF_{T_F}(w_i), WBG_{T_G}(w_i), \langle \mathrm{id}_f, \mathbf{0} \rangle \oplus H_1\left(WBP_{T_P}(w_i), r_i\right), \\ r_i, \langle \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, WBF_{T_F}(w_i) \rangle \oplus H_2\left(WBP_{T_L}(\mathrm{id}_f), r_i'\right), \mathrm{r_i'} \end{pmatrix}$$

  where r_i and r_i' are random k-bit strings.
  3. Compute $c_f \leftarrow \mathrm{WBEnc}_{T_{Enc}}(f)$
  4. Output (t_a, c_f)

- $\mathrm{DelToken}(\boldsymbol{K}, fid)$: using the master keys $\boldsymbol{K}$ and the file ID fid outputs a token for deleting the file with that ID.

$$t_d := \left(R_{K_5}(fid), M_{K_6}(fid), L_{K_7}(fid), fid\right)$$

- $\mathrm{GatewayDelToken}(\boldsymbol{T}_u, fid)$: using the update gateway $\boldsymbol{T}_u \stackrel{\text{def}}{=} (T_F, T_G, T_P, T_R, T_M, T_L, T_{\mathrm{Enc}})$ and the file id fid returns the following token:

$$t_d := \left(\mathrm{WBR}_{T_R}(fid), \mathrm{WBM}_{T_M}(fid), \mathrm{WBL}_{T_L}(fid), fid\right)$$

As can be noticed the resulted deletion token is similar to the token generated by the $\mathrm{DelToken}(\boldsymbol{K}, fid)$ algorithm for the same file ID.

- $\mathrm{Add}(\gamma, \boldsymbol{c}, t_a)$: using the addition token generated either by the key owner or by the eligible third-party user updates the secure index in a following way:
  a. Parse t_a as $(t_1, t_2, \lambda_1, \ldots, \lambda_l, c)$ and $t_1 \in T_d$ return $Error[FileAlreadyExists]$, otherwise:
  b. For $1 \leq i \leq l$,
     1. Compute $(\xi, \mathbf{0}) := T_s[free]$. ξ is the last unused location in A_s and find the ξ^* pair of ξ in the A_s array by getting $(\xi_{-1}, \xi^*) := A_s[\xi]$.
     2. Update the value for $free$ word in T_s table to point to the penultimate free node:
     $$T_s[free] := \left(\xi_{-1}, \mathbf{0}^{\log|A_s|}\right)$$
     3. Each λ_i for $i \in \{1, \ldots, l\}$ refers to a single keyword w_{λ_i} of the new file. So for each i do the following.
        i. Reveal address of the $L_{w_{\lambda_i}}$ list's first node N_1
        $$(p_1, p_1^*) := T_s[\lambda_i[1]] \oplus \lambda_i[2]$$
        ii. Create a new node at the location ξ of A_s array and make it to point to the node N_1 by setting:
        $$A_s[\xi] := (\lambda_i[3] \oplus \langle \mathbf{0}, p_1 \rangle, \lambda_i[4])$$
        iii. Update T_s by setting:
        $$T_s[\lambda_i[1]] := (\xi, \xi^*) \oplus t_2$$
        iv. Update the pair node of N_1 by setting
        $$A_d[p_1^*] := (D_1 \oplus \langle \mathbf{0}, \xi^*, \mathbf{0}, \mathbf{0}, \xi, \mathbf{0}, \mathbf{0} \rangle, r)$$
        where $(D_1, r) := A_d[p_1^*]$
        v. Update the dual of $A_s[\xi]$ by setting
        $$A_d[\xi^*] := (\lambda_i[5] \oplus \langle \xi_{-1}^*, \mathbf{0}, p_1^*, \xi, \mathbf{0}, p_1, \lambda_i[1] \rangle, \lambda_i[6])$$
        vi. In the first iteration of the loop (if $i = 1$) update the T_d table:
        $$T_d[t_1] := \xi^* \oplus t_2$$
  c. When the loop finished the secure index is updated so update the ciphertexts collection by adding c to $\boldsymbol{c}$.

- $\mathrm{Del}(\gamma, \boldsymbol{c}, t_d)$:
  a. Parse t_d as (t_1, t_2, t_3, fid) and if $t_1 \notin T_d$ return $Error[FileNotFound]$, otherwise:
  b. Find L_f list's first node:

$$p'_1 := T_d[t_1] \oplus t_2$$

  c. Start a loop with $i := 1$,
    1. Parse $(D_i, r) := A_d[p'_i]$ and compute $(a_1, \ldots, a_6, \mu) := D_i \oplus H_2(t_3, r)$.
    2. Delete D_i by setting $A_d[p'_i]$ equal to a pair of random strings of $(6 * \log|A_d| + k; k)$-bits length.
    3. Find the pointer to the last unused node by getting $\left(\xi, \mathbf{0}^{\log|A_s|}\right) := T_s[free]$
    4. Increase the number of the unused nodes in the search array by one (the deleted one) by setting:

$$T_s[free] := \langle a_4, \mathbf{0}^{\log|A_s|} \rangle$$

    5. Free the location of the D_i's dual by setting $A_s[a_4] := (\xi, p'_i)$
    6. Let N_{-1} be the previous node of D_i's dual. Update the node which comes after N_{-1} by setting:

$$A_s[a_5] := (b_1, b_2 \oplus a_4 \oplus a_6, r_{-1})$$

    where $(b_1, b_2, r_{-1}) := A_s[a_5]$. Also update the pointers of the pair node of N_{-1} by setting:

$$A_d[a_2] := (b_1, b_2, b_3 \oplus p'_i \oplus a_3, b_4, b_5, b_6 \oplus a_4 \oplus a_6, \mu^*, r^*_{-1}) ,$$

    Where $(b_1, \ldots, b_6, \mu^*, r^*_{-1}) := A_d[a_2]$
    7. let N_{+1} be the node that follows D_i's dual. Update N_{+1}'s pair's pointer:

$$A_d[a_3] := (b_1, b_2 \oplus p'_i \oplus a_2, b_3, b_4, b_5 \oplus a_4 \oplus a_5, b_6, \mu^*, r^*_{+1})$$

    where $(b_1, \ldots, b_6, \mu^*, r^*_{+1}) := A_d[a_3]$
    8. Set $p'_{i+1} := a_1$.
  d. Remove the ciphertext for file with fid ID from $\boldsymbol{c}$.
  e. Remove t_1 from T_d
- $\mathrm{Dec}(\boldsymbol{K}, c)$: For a single ciphertext c output $f := \mathrm{SKE.Dec}_{K_4}(c)$.

As can be seen the gateways generated for third-party users are based on the white box tables corresponding to the secret keys. Using this construction the key owner can grant the third-parties either search permission or both search and update permissions. Having only the search gateway a third-party user cannot run modification queries (addition and deletion).

Using the same method most of the existing single-user searchable encryption schemes can be extended for supporting third-party user's access to the data via search and/or modification queries.

2.3 SECURITY

There is no formal framework developed for describing a multi-party SE scheme's security. Intuitively such a scheme should provide data security and secret key security against malicious third-party users and the untrusted server. In all SE schemes the server is considered as a possible passive adversary which is interested in obtaining more information about the user's data, however it never acts maliciously or interacts with malicious users. The scheme described in [**65**] does not provide key secrecy at all. The master key used for secure index construction is shared with all eligible third-parties. This means a malicious third-party user is able to decrypt all secret information by hacking the server's database. The latest scheme described in [**69**] provides key secrecy at the cost of additional communication round per query between the third-party and the key owner.

In our setting we also assume that the server is honest-but-curious which never acts maliciously. Otherwise a malicious server can always act maliciously even without knowing the secret information. For example it can return incorrect results to the search queries. This consideration of honest-but-curious server is quite practical in real-life situations, allows the owner of data to configure an access-control mechanism on server and control the users' permissions.

We specify the following security aspects:

1. Secret key's security against third-party users.

2. Data security against revoked third-party users
3. Data security against the honest-but-curious server.
4. Secret key security against honest-but-curious server

Proof Sketch #1: The third-party users are provided with the WB encryption tables $T_F, T_G, T_P, T_R, T_M, T_L, T_{\text{Enc}}$ which hide the keys $K_1, K_2, K_3, K_4, K_5, K_6, K_7$. The secret key security against third-party users is guaranteed by the security of WB table generation algorithm against key recovery attacks. If the underlying WB encryption scheme is not resistant against key-recovery attacks, the third-parties will be able to extract the secret keys from the tables. This will open lots of security issues against the system's overall security.

Proof Sketch #2: The data security against revoked third-party users is guaranteed by the access control mechanism configured by the owner of the data on the server. The revoked users should be rejected access by the server. In case the revoked user manages to access the encrypted database (by hacking or stealing the database server) it will be able to run queries and get the results though it wouldn't be able to decrypt the index and get plaintext information. This means that if the attacker is wondered in the content of specific file, he can only make guesses which words can exist in that file and then check by making query over the index. Finding a word he will never learn the position of the word or the context of its usage. This is a significant security progress.

Proof Sketch #3: Data security against the honest-but-curious server is the most important security aspect in the SE scheme. The server given the encrypted index $\gamma \stackrel{\text{def}}{=} (A_s, A_d, T_s, T_d)$ and the set of ciphertexts $\boldsymbol{c} \stackrel{\text{def}}{=} (c_1, \ldots, c_n)$ should never learn any private information regarding these files.

All practical SE schemes which are not based on Oblivious RAM [**77**], [**76**] approaches leak some information. Intuitively a dynamic scheme should leak more information than a static scheme or a scheme which provides lesser level of dynamicity. In [**17**] scheme a formal framework is provided for describing the scheme's leakage and a formal security proof. Here it is shown that our multiuser scheme does not increase the system's leakage and provides the same security level as the scheme from that paper.

Next the formal definitions of the leakage functions $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ which altogether describe the overall leakage of the scheme are provided.

- The $\mathcal{L}_1$ leakage is described as

$$\mathcal{L}_1(\boldsymbol{f}) \stackrel{\text{def}}{=} \left(|A_s|, \{\mathrm{id}(w): \exists f \in \boldsymbol{f}; w \in W_f\}, \{\mathrm{id}(f)\}_{f\in\boldsymbol{f}}, \{|f|\}_{f\in\boldsymbol{f}}\right)$$

  Here id is a function returning the unique ID of the word or the file. This function shows the leakage of the search array size, the IDs of keywords and their total number, the IDs of files and their total number and the size of each file.

- The $\mathcal{L}_2$ leakage is described as

$$\mathcal{L}_2(\boldsymbol{f}, w) \stackrel{\text{def}}{=} (\mathrm{id}(w), \mathrm{accpt}(w))$$

  This function describes the leakage of each word ID as well as access pattern which is the IDs of all files containing the specified word w.

- The $\mathcal{L}_3$ leakage is described as

$$\mathcal{L}_3(\boldsymbol{f}, f) \stackrel{\text{def}}{=} (\mathrm{id}(f), \{\mathrm{id}(w), \mathrm{apprs}(w)\}_{w\in W}, |f|)$$

  Here $\mathrm{apprs}(w)$ a Boolean with a value equal to 1 if w at least appears in one of the existing files $\boldsymbol{f}$, and with value 0 value otherwise. This function shows that having a new file, the IDs of each unique word contained in that file and the appearance bit of the word are revealed.

- The $\mathcal{L}_4$ leakage is described as

$$\mathcal{L}_4(\boldsymbol{f}, fid) \stackrel{\text{def}}{=} (fid, \{\mathrm{id}(w), \mathrm{prev}(fid, w), \mathrm{next}(fid, w)\}_{w\in W})$$

  Here $\mathrm{prev}(fid, w)$ and $\mathrm{next}(fid, w)$ are the identities of the files before and after the files with fid ID that contain w. If there are no files before or after fid that contain the word then $\mathrm{prev}(fid, w)$ and $\mathrm{next}(fid, w)$ return $NULL$. This function shows that file deletion reveals the identities of the words contained in the file as well as the succeeding and preceding file identifiers containing the selected word.

Let A be a stateful adversary, SSE be the dynamic SE scheme described in the previous section, S be again a stateful simulator and $\boldsymbol{\mathcal{L}} \stackrel{\text{def}}{=} (\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ be the leakage functions as they were described above. While discussing the data security let's consider the following experiments:

$\mathbf{Real}_{\boldsymbol{A}}(k)$: The challenger runs $\mathrm{Gen}(k)$ to generate master keys $\boldsymbol{K}$ then the adversary A sends to the challenger a set of files $\boldsymbol{f}$ and receives pair $(\gamma, \boldsymbol{c}) \leftarrow \mathrm{Enc}_K(\boldsymbol{f})$ from it. After this A creates l adaptive queries and sends to the challenger. Then for each query $q \stackrel{\text{def}}{=} \{w, f, fid\}$ the challenger sends to the adversary one of the search $t_s \leftarrow \mathrm{SrchToken}(w, K_1, K_2, K_3)$; addition $(t_a, c_f) \leftarrow \mathrm{AddToken}(\boldsymbol{K}, f)$ or deletion $t_d \leftarrow \mathrm{DelToken}(\boldsymbol{K}, fid)$ tokens, for addition token it also sends the ciphertext of the file which is being added. As a result of experiment A returns a bit b.

$\mathbf{Ideal}_{\boldsymbol{A},\boldsymbol{S}}(k)$: The adversary A sends to the simulator S a set of files $\boldsymbol{f}$. Given $\mathcal{L}_1(\boldsymbol{f})$ the simulator S generates and sends a pair $(\gamma, \boldsymbol{c})$ to A. After this A creates l adaptive queries and sends to the simulator. Then for each query $q \stackrel{\text{def}}{=} \{w, f, fid\}$ the simulator sends to the adversary one of the search t_s; addition (t_a, c_{f_1}) or deletion t_d tokens, for addition token it also sends the ciphertext of the file which is being added. To generate search token the simulator uses $\mathcal{L}_2(\boldsymbol{f}, w)$, for addition token it uses $\mathcal{L}_3(\boldsymbol{f}, f)$ and for deletion token $\mathcal{L}_4(\boldsymbol{f}, fid)$. Here again as a result of experiment the adversary returns a bit b.

Definition 2.1 (dynamic CKA2-security): The SSE is *$\mathcal{L}$-secure against adaptive and dynamic chosen-keyword attacks* if for all probabilistic polynomial-time (PPT) adversaries A, there exists a PPT simulator S such that

$$\left|\Pr\left[\mathbf{Ideal}_{\boldsymbol{A},\boldsymbol{S}}(k) = 1\right] - \Pr[\mathbf{Real}_{\boldsymbol{A}}(k) = 1]\right| < \mathrm{negl}(k)$$

The security of a multi-user scheme can be defined similarly to the security of a single-user scheme, as the server should not learn anything about the documents and queries beyond what can be inferred from the access and search patterns. The main security theorem exposing the security of the scheme is defined as follows.

Theorem 2.1: The multiuser SE scheme described as a tuple of 14 algorithms SSE = (Gen, Enc, GenSearchGateway, GenUpdateGateway, SearchToken, GatewaySearchToken, AddToken, GatewayAddToken, DelToken, GatewayDelToken, Search, Add, Del, Dec) is $\mathcal{L}$-secure against adaptive and dynamic chosen-keyword attacks given a random oracles H_1 and H_2, PRFs F, G, P, R, M and L, secure white-box encryption algorithms corresponding to that PRFs and a CPA-secure SYM encryption scheme.

The security defined by this theorem claims that given a sequence of search, deletion or addition tokens $\boldsymbol{t} \stackrel{\text{def}}{=} (t_1, \ldots, t_l)$ for the queries $\boldsymbol{q} \stackrel{\text{def}}{=} (q_1, \ldots, q_l)$ generated adaptively the server should not learn more information about either $\boldsymbol{f}$ than defined by the leakage functions $\boldsymbol{\mathcal{L}}$.

From the server's point of view our settings differs from the single user configuration by the fact that besides the data owner the tokens can be generated by the third-party users.

Suppose that the key owner generated a search token $t_s := \left(P_{K_3}(w), G_{K_2}(w), F_{K_1}(w)\right)$ and the third-party's generated search token is $t_s' := \left(WBP_{T_P}(w), WBG_{T_G}(w), WBF_{T_F}(w)\right)$. As WB scheme encryption with the WB tables results to the same ciphertext which is obtained by regular key encryption since $WB\$_{T_\$}(w) \equiv \$_{K_\%}(w)$ for $(\$, \%) \in \{(F,1),(G,2),(P,3)\}$ and $T_\$ = \mathrm{WBGen}(\$, K_\%)$, then the tokens t_s' and t_s are the same. This means the server does not get anything more in the multi-user setup than it gets in the single-user setting and the third-party generated tokens are indistinguishable from the key-owner's generated tokens. The same situation is in the case of addition and deletion tokens. Thus the security against the server is the same as in the single–user scheme.

At high level this theorem is proved in the following way. The simulator S simulates the pair $(\tilde{\gamma}, \tilde{\boldsymbol{c}})$ of secure index and set of file ciphertext getting as an input the set of file $\boldsymbol{f}$ and using $\mathcal{L}_1(\boldsymbol{f})$. $\mathcal{L}_1$ returns the number of files, their sizes and ids and word ids and the size of the search array A_s. The generated index $\tilde{\gamma}$ can be constructed similarly to a real secure index γ, expect some actions. Namely outputs of the PRFs of the scheme are changed by random values and all encryptions are done on strings of 0-s of appropriate length. The fact that the encryption schemes are CPA-secure and functions F, G, P, R, M and L are pseudo-random will ensure that $\tilde{\gamma}$ is indistinguishable from γ generated during real execution. The simulated collection of file ciphertexts $\tilde{\boldsymbol{c}}$ is generated in the same way and again encryption scheme's CPA-security guarantees the indistinguishability.

In the same manner the simulated tokens should be generated for search and update operations from the information received from $\mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4$. However simulation of the tokens

is more complex, because of keeping track of various dependences revealed during the previous queries of the simulator.

We do not expose the full proof here which is shown in [**17**], as nothing will be changed in the formal description related to the setting changes. As the tokens generated either by the key-owner or third-party users are indistinguishable, the security proof remains the same.

Proof Sketch #4: Third-party users are provided with the WB encryption tables $T_F, T_G, T_P, T_R, T_M, T_L, T_{\text{Enc}}$ which hide the keys $K_1, K_2, K_3, K_4, K_5, K_6, K_7$. There is a risk that some malicious third-party user will cooperate and share with the server his WB encryption tables. In this case the server will be able to query the database as usual trusted user and get statistical information about the database. He will be able also to generate the corresponding tokens for each word. Having the history of queries, the server will be able to reveal which queries had been made in the past by the users. Here again the secret key security will be guaranteed by the security WB encryption scheme. If the WB scheme is resistant against the key-recovery attacks, the secret key will not be revealed.

2.4 IMPLEMENTATION

To demonstrate our algorithms we implemented a C++ library and a demo application which demonstrates the usage of that library. The library is named DSSE (dynamic searchable symmetric encryption) and provides API for programming both client and server side applications.

The main class of the client side API is QueryTokenGenerator (see Fig: 2.2), which provide the ability to create objects of SearchToken, DeleteToken (see Fig: 2.3) and AddToken (Fig: 2.4) types.

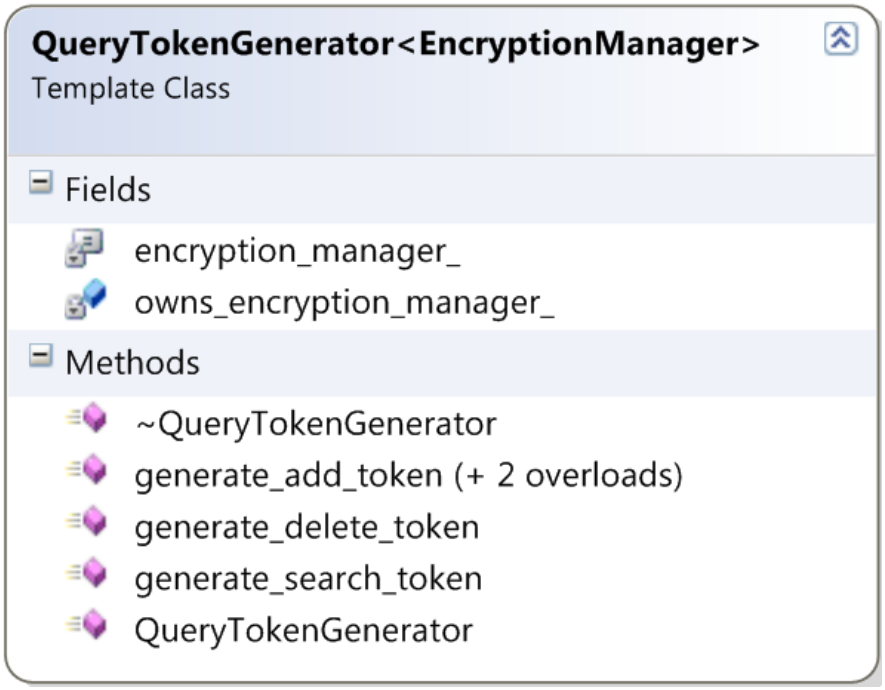
QueryTokenGenerator<EncryptionManager>
Template Class
Fields
encryption_manager_
owns_encryption_manager_
Methods
~QueryTokenGenerator
generate_add_token (+ 2 overloads)
generate_delete_token
generate_search_token
QueryTokenGenerator

Fig: 2.2

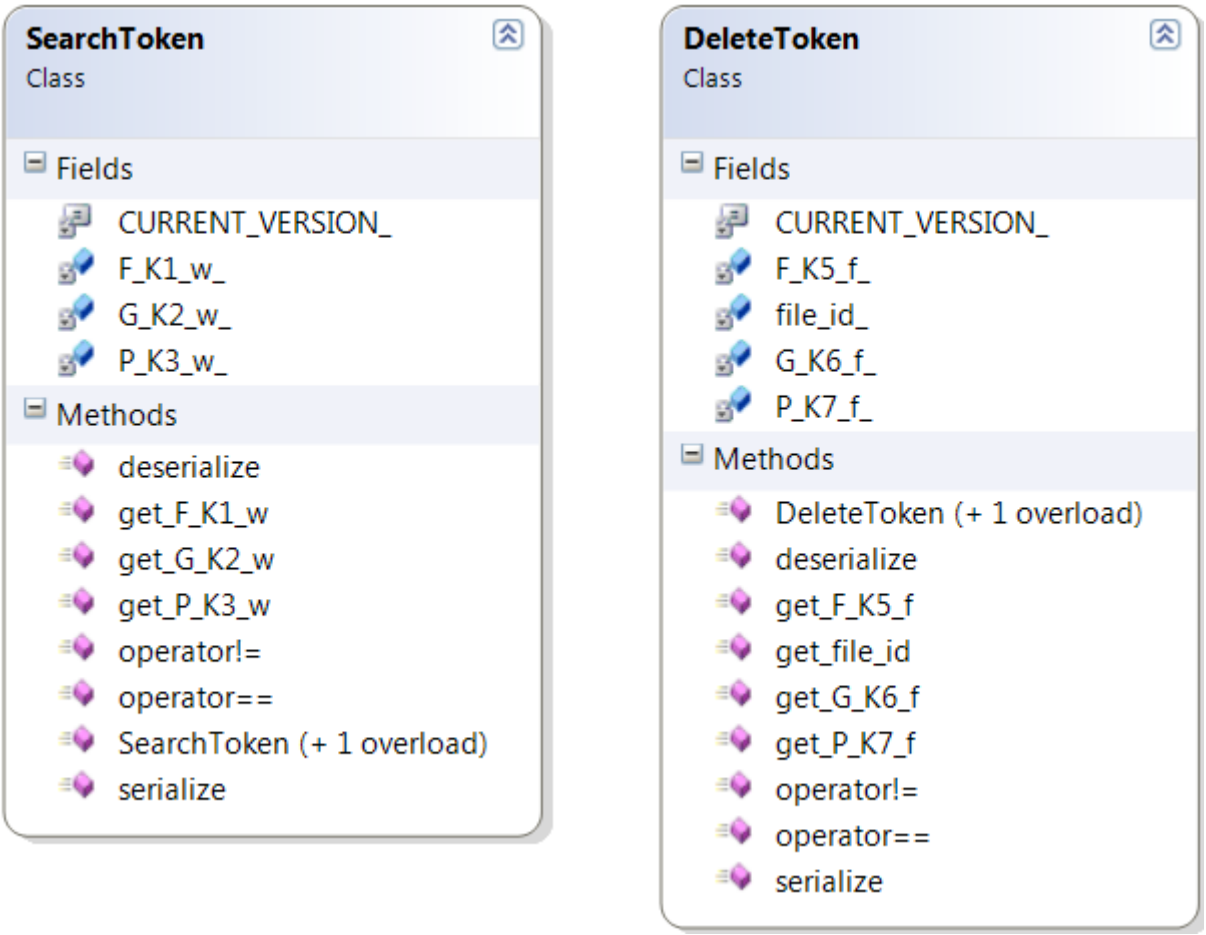
SearchToken
Class
Fields
CURRENT_VERSION_
F_K1_w_
G_K2_w_
P_K3_w_
Methods
deserialize
get_F_K1_w
get_G_K2_w
get_P_K3_w
operator!=
operator==
SearchToken (+ 1 overload)
serialize
DeleteToken
Class
Fields
CURRENT_VERSION_
F_K5_f_
file_id_
G_K6_f_
P_K7_f_
Methods
DeleteToken (+ 1 overload)
deserialize
get_F_K5_f
get_file_id
get_G_K6_f
get_P_K7_f
operator!=
operator==
serialize

Fig: 2.3

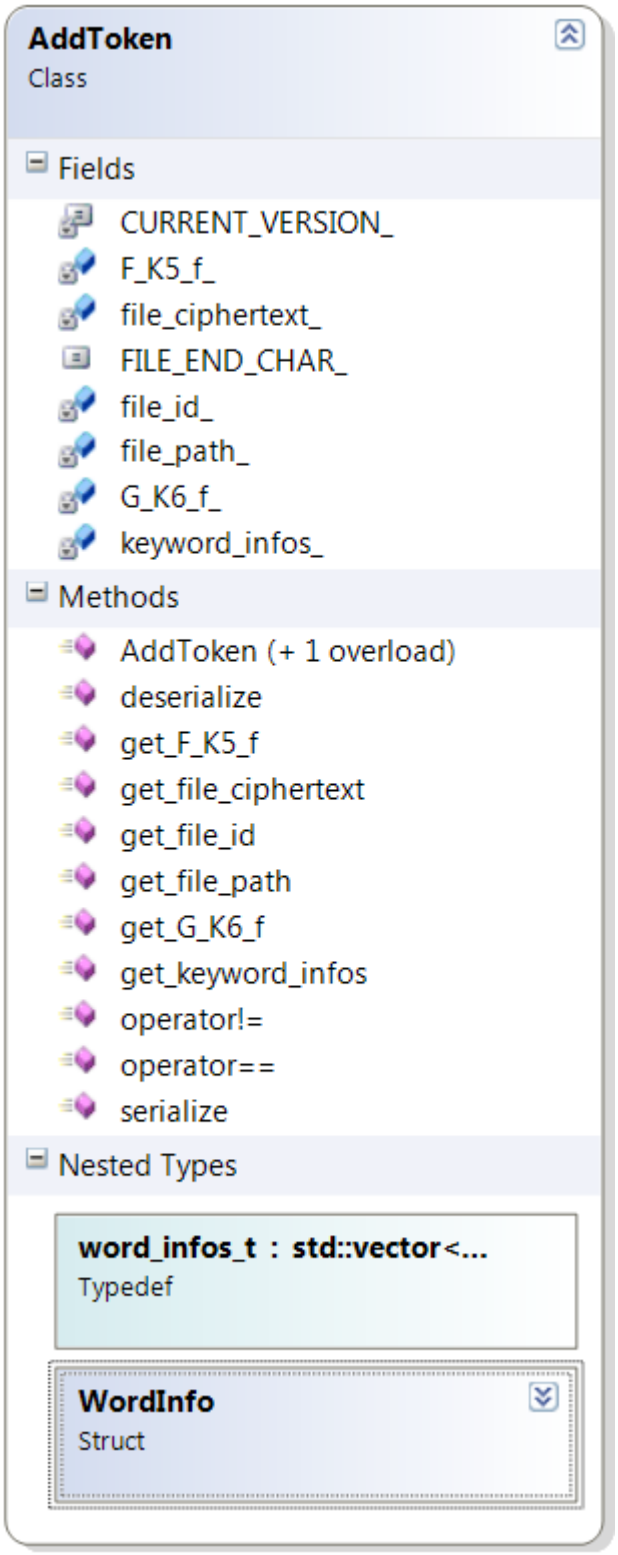

Fig: 2.4

Each of these token classes represents corresponding token object described in the algorithm. All of the token classes have serialize() and deserialize() methods intended to be used for the token's transmission via socket.

For creation of instance of the QueryTokenGenerator type itself an encryption manager should be used. The encryption manager is responsible for providing seven encryption boxes for the master keys $K_1 \dots K_7$ of the scheme. As it is a part of the key management we provide an API for implementing custom encryption manager, namely an abstract class EncryptionManagerBase (see Fig: 2.5)

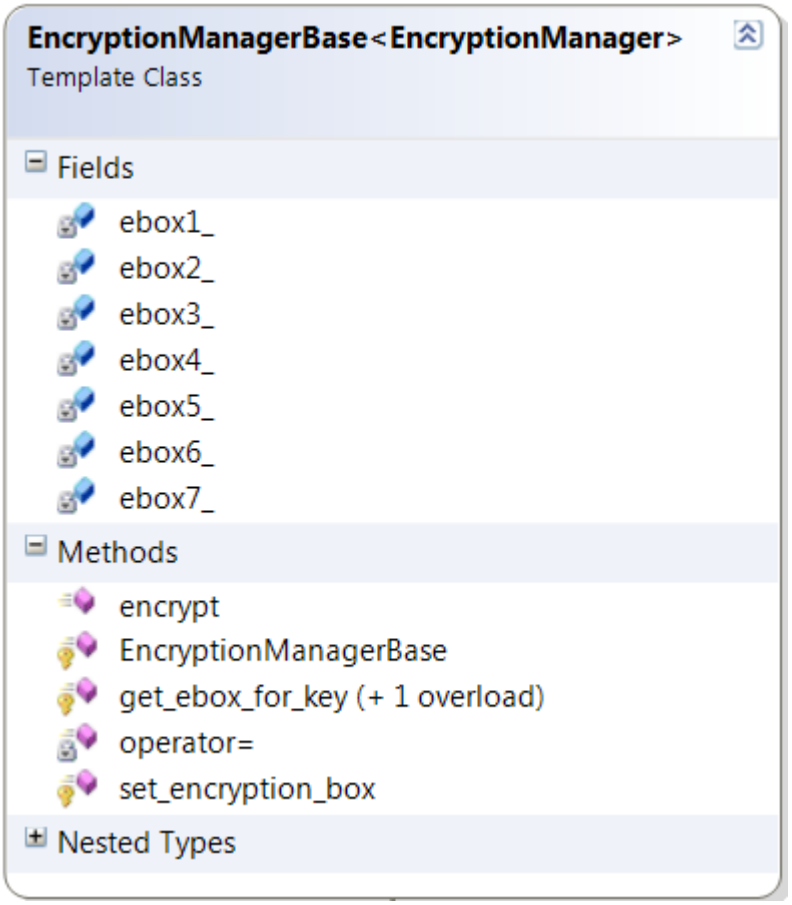

Fig: 2.5

Also we provide two implementations EncryptionManagerSaferPlusBlackBox (see Fig: 2.6) and EncryptionManagerSaferPlusWhiteBox (see Fig: 2.7) of the encryption manager interface. First of which can be used by the data owner and the second one by the third-party users.

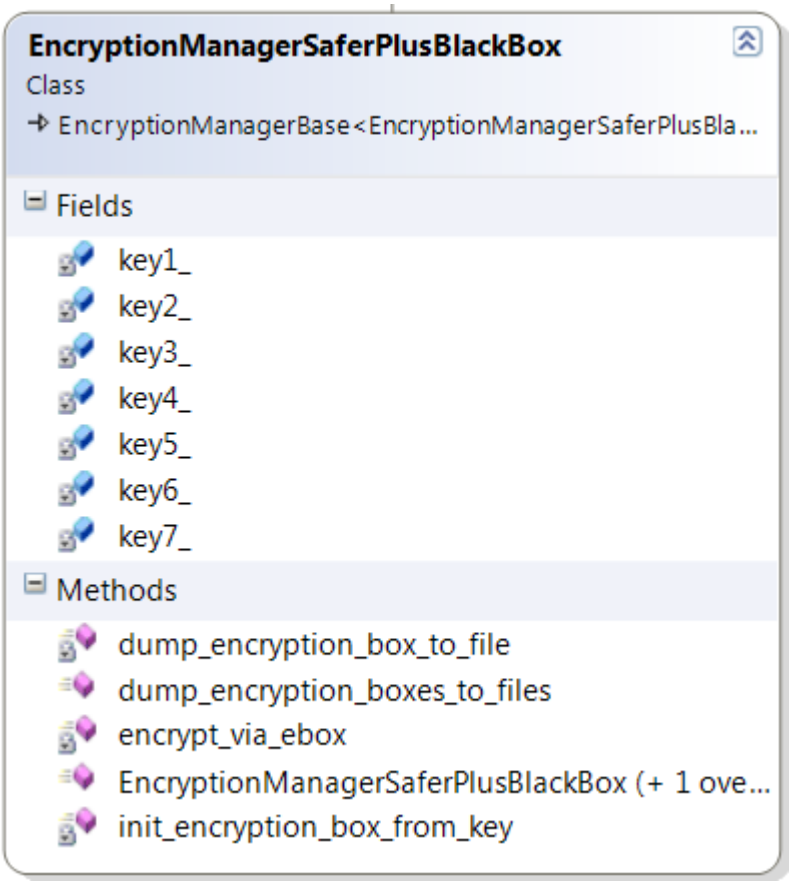

Fig: 2.6

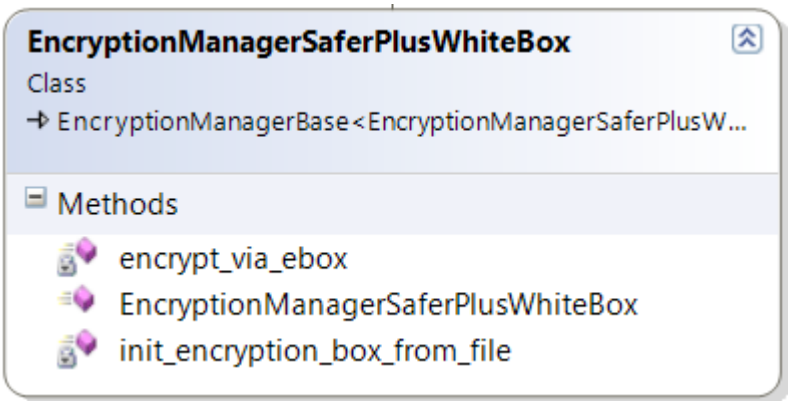

Fig: 2.7

The sever side API's main class is FileDbManager (see Fig: 2.8). This class manages the database of files. It provides ability to search, add, delete files from the database using corresponding search, add, delete tokens and custom implementations for database index and database data. To create an instance of FileDbManager class the directory of file db should be specified. Under that directory a directory "data" is being created for storing encrypted files, and a file "index" for storing encrypted index of the file database.

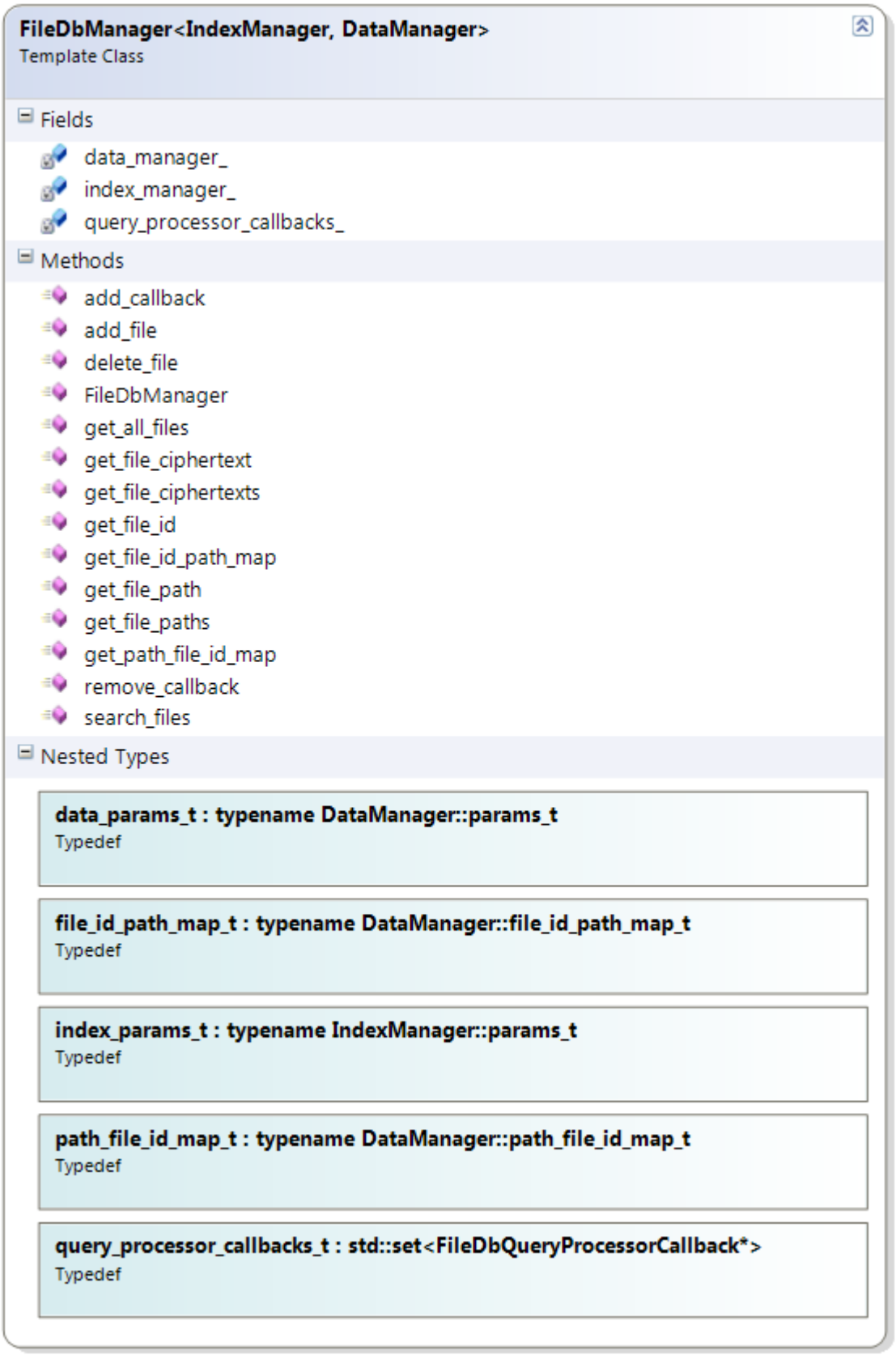
FileDbManager<IndexManager, DataManager>
Template Class
Fields
data_manager_
index_manager_
query_processor_callbacks_
Methods
add_callback
add_file
delete_file
FileDbManager
get_all_files
get_file_ciphertext
get_file_ciphertexts
get_file_id
get_file_id_path_map
get_file_path
get_file_paths
get_path_file_id_map
remove_callback
search_files
Nested Types
data_params_t : typename DataManager::params_t
Typedef
file_id_path_map_t : typename DataManager::file_id_path_map_t
Typedef
index_params_t : typename IndexManager::params_t
Typedef
path_file_id_map_t : typename DataManager::path_file_id_map_t
Typedef
query_processor_callbacks_t : std::set<FileDbQueryProcessorCallback*>
Typedef

Fig: 2.8

For processing (reading, writing) the secure index which is done by the IndexManager parameter of FileDbManager class two classes (FileDbIndexManagerMappedFile and FileDbIndexManagerInMemory) have been implemented. First of them stores the index in a memory mapped file so it is stateful hence more practical, but the second one is only for testing purposes since it gets deleted after application is finished (see Fig: 2.9).

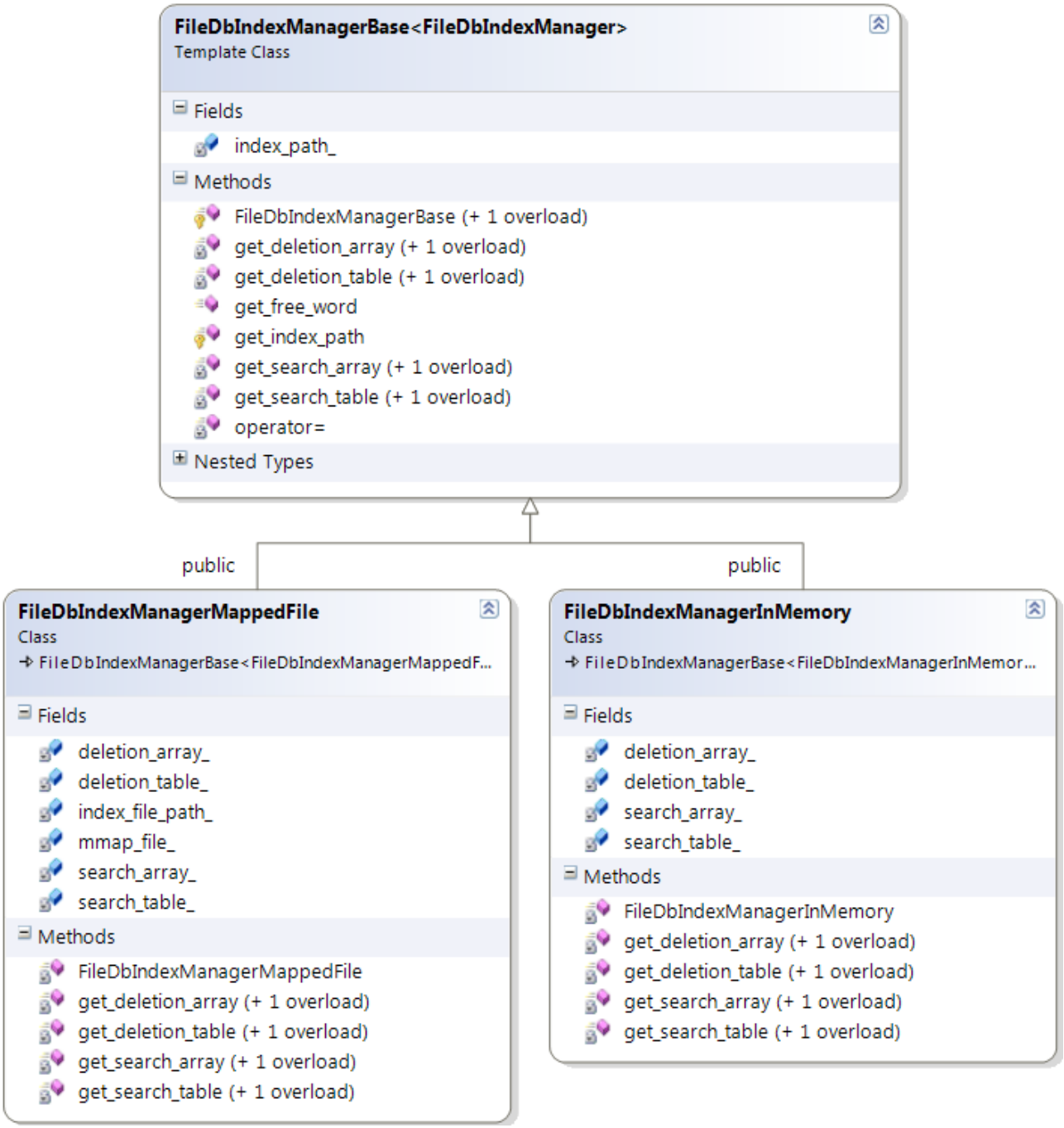

Fig: 2.9

The demo QT based GUI application conducts both client and server side operations. At the beginning it creates an empty file database then allows running queries against that database from console. It allows to:

- Add a file or a directory to the database.
- Delete a file or a directory from the database.
- Search files with specified keywords.

The cryptographic primitives for our implementation are 128-bit SAFER+ [**88**] block cipher algorithm and SHA-256 for hashing. The WB encryption scheme is based on the technology described in [**90**].

During the implementation the following libraries are used:

- Boost Libraries version 1.55.0. (bind, lexical_cast, uuid and interprocess).
- Implementation of SHA256 and SHA224 by from Crypto++ library.
- Implementation of WB encryption algorithm based on 128-bit SAFER+.

CHAPTER 3. IMPLEMENTATION OF SEARCH FUNCTIONALITY OVER ENCRYPTED CLOUD DATA

Dropbox [**6**], Google Drive [**7**], Box [**8**], Microsoft OneDrive [**9**] or any other public cloud storage provider stores and can read all information users store there. This is a very serious security issue [**10**] and for all individuals and organizations that cares about the security of their data but still wants to benefit from public cloud storages the only solution is to use some encryption tool which will help to encrypt user information before uploading it to the cloud. The design of advanced security solution for public cloud storages and for distributed file storages in general is a hard scientific and technical problem [**95**], [**96**], [**97**], [**98**], [**99**], [**100**], [**67**]. On the other hand, there is a tradeoff between security and usability, as encryption eliminates the easy access to data via search and also makes the sharing and collaboration harder. Various special cloud encryption gateways had been developed in recent years aiming to secure the users data in public cloud storages without compromising the sharing and collaboration features provided by the storage providers. In this chapter we review the main solutions existing in this domain showing what level of security is provided by each of them and their main advantages and disadvantages from both the security and usability points of view. It is important that none of existing cloud encryption gateways provide search functionality over encrypted data. Next we present our own cloud encryption architecture which was embedded into a new cloud securing system called SkyCryptor. It provides very high level security for users and implements search functionality over encrypted data.

3.1 CLOUD ENCRYPTION GETWAYS

There are dozens of cloud encryption tools operating in the market. In this section we review the most popular solutions.

3.1.1 SOOKASA

Sookasa [**55**] is a new emerged cloud encryption gateway specially designed to help the companies from regulated industries and facilitate compliance with six federal standards such as HIPAA, FERPA, PCI DSS, GLBA, FINRA, SOX. It helps such organizations to store their data in cloud in encrypted form and also have full visibility on how the data was used or shared. However Sookasa does not provide a perfect secrecy as it handles all user secret key management on behalf of the users.

Sookasa secures the users files in Dropbox in the following manner:

1. Sookasa creates a special folder in user's Dropbox.
2. The user put sensitive files in that folder which are seamlessly encrypted with AES-256 encryption [**101**] with unique file key randomly generated for that file.
3. Sookasa encrypts the key for file encryption with the Sookasa's public key. The key of an encrypted file is stored at the beginning of the file.
4. The encrypted files are synced among all devices
5. When Alice shares some file with Bob and Bob wants to access Alice's encrypted file, Sookasa's server takes the file's key encrypted with the public key of the server, decrypts it and sends the key of the file to Bob.

As can be seen Sookasa owns all encryption keys used for securing the users files. This is a serious security drawback as the powerful adversary or Sookasa itself can always access the users sensitive files Such solution may satisfy specific companies but it cannot be a reliable security solution for companies which want to fully exclude the chance of their data appearing into third-parties hands.

3.1.2 NCRYPTEDCLOUD

nCryptedCloud [**54**] is another cloud encryption gateway working with most cloud storage providers such as Dropbox, Google Drive, OneDrive and Egnyte. It provides a rich functionality of file/folder sharing and unlike Sookasa allows securing any file in any folder. However from the security point of view there is still a little difference between Sookasa and

nCryptedCloud. The later provide perfect security for individual files meaning the user does not need to share the keys for file encryption with nCryptedCloud as far as the file is not required to be shared with other users. But for securely collaborating on cloud files, the user again needs to share the keys for encrypting files with nCryptedCloud's server. The following list highlights the main file storing and sharing functionality.

File encryption works as follows:

1. Alice creates a secure unique password for her file.
2. Alice encrypts the plaintext data using AES-256 Zip encryption by using the generated password.
3. Alice encrypts the file's password using her public key and stores the encrypted password of the encrypted Zip file on the server somewhere next to the encrypted file.
4. When Alice need to share the file with Bob, she encrypts the file's password with nCryptedCloud's server's public key and sends it also sends to the server.
5. When she wants to open the encrypted file, she just takes the encrypted password from the zipped file and decrypts that using her private key. Next she uses that password to decrypt the file encrypted by AES.
6. Bob receives the shared file and when he wants to access that files, nCryptedCloud verifies that Bob has access to the file key and distributes it to him. Bob stores the received key on his local store of the keys, so for further accesses he does not need the nCryptedCloud to distribute him the file key.

Again the main drawback of nCryptedCloud is the fact that it can learn the secret keys and/or passwords used for encrypting files. Although they claim that they never can access the cloud encrypted files, theoretically they can do it having the cloud storage access token for each user as well as the secret user generated keys for securing data.

3.1.3 BOXCRYPTOR

Boxcryptor [**56**] is the only gateway for cloud encryption among the existing solutions providing a zero-knowledge service to users. Its secure key management is based on

asymmetric RSA cryptosystem and here files are encrypted with AES-256 block cipher. Each user has own pair of private and public keys. The file encryption procedure in the non-corporate case works as follows:

1. Randomly create a secure file key.
2. Then encrypt the file using its key.
3. Using the public key of the user encrypt the file key.
4. Next to the ciphertext of the file append its file key.
5. In case when the file is shared among multiple users, encrypt its key using the public key of each permitted user and append those ciphertext to the encrypted file.

The main drawback of Boxcryptor is the fact, that if the file is accessible for multiple users then its key is encrypted as much times as the number of users and each of the results is appended to the encrypted file. This causes the file to get re-encrypted every time when the group of people having access to the file is changed. Also the file size is growing linearly with number of people having access to it as for each new user having access to that file a new ciphertext should be stored at the beginning of the file.

3.1.4 SKYCRYPTOR

SkyCryptor [**59**] is a novel cloud encryption gateway which goal is to provide zero-knowledge security to users by preserving the main advantages of cloud storage services. Its key management technique is based on so called proxy re-encryption scheme [**102**], [**103**], [**104**] in which context the SkyCryptor service acts as a semi-trusted proxy server responsible for keeping proxy re-encryption keys and re-encrypting the file encryption keys upon authorized access request. Each user has a pair of public and private keys specific to the proxy public key encryption algorithm. The encryption of a file works as follows:

1. For each file Alice generates a random key for file's encryption and encrypts the file with AES-256 CBC mode.

2. Alice encrypts the file's encryption key with her public key via proxy public key encryption algorithm. The key of the encrypted file is appended to the encrypted file and is stored in the cloud.
3. When Alice wants make her file accessible for Bob, she creates a special proxy re-encryption key and stores it in SkyCryptor servers. Alice also creates permission for Bob allowing him to access the file.
4. When Bob wants to decrypt the Alice's shared file, he takes the file key encrypted with Alice's public key from the cloud-stored file and sends it to SkyCryptor service. SkyCryptor checks the permission and re-encrypts the ciphertext with help of the proxy re-encryption key so the result is already the file key encrypted with Bob's public key. The result is sent to Bob.
5. Bob receives the encrypted file key, decrypts it with own private key and reveals the key, which can be used finally to decrypt the encrypted file.

As can be seen, SkyCryptor server never learns the file keys. The proxy re-encryption allows re-encrypting the ciphertext without decrypting them. This powerful technique allows building an efficient and privacy-preserving cloud encryption gateway.

The next fundamental advantage of SkyCryptor is that it provides search functionality over encrypted data based on proprietary searchable encryption algorithm.

3.2 SEARCHABLE ENCRYPTION IMPLEMENTATION ASPECTS

Providing zero-knowledge search functionality over encrypted data means that the host server which keeps the user's data and/or the searchable index, should not learn any information about the file content at any moment of its operation. This means that the files, searchable indexes or even search queries never should appear on servers in the plaintext form. Among these security requirements the following issues should be considered while implementing a secure indexing and search functionality for large-scale application.

- The user can add files from different devices.
- The user should be able to search from any of his devices or web application.

- The user should have access only to his files.

The searchable encryption algorithm used by SkyCryptor requires several AES-256 keys to be employed during secure index construction. Each user has specific searchable encryption master key and all other keys required for index construction or search are generated from the master searchable encryption key. As the user should be able to make search from both web service and client applications, the master key should be accessible on all this platforms. But from the other hand it should never be revealed at server side. The master key generation and recovery is done as follows:

1. When the user first login via some client application, he generates the searchable encryption master key among other encryption keys.
2. The user encrypts the searchable encryption master key with his password-key. The password-key is an AES-256 key generated from the user's password with help of key stretching algorithm PBKDF2, which operates with HMAC-256 and 50000 iterations and the username is taken as a salt value.

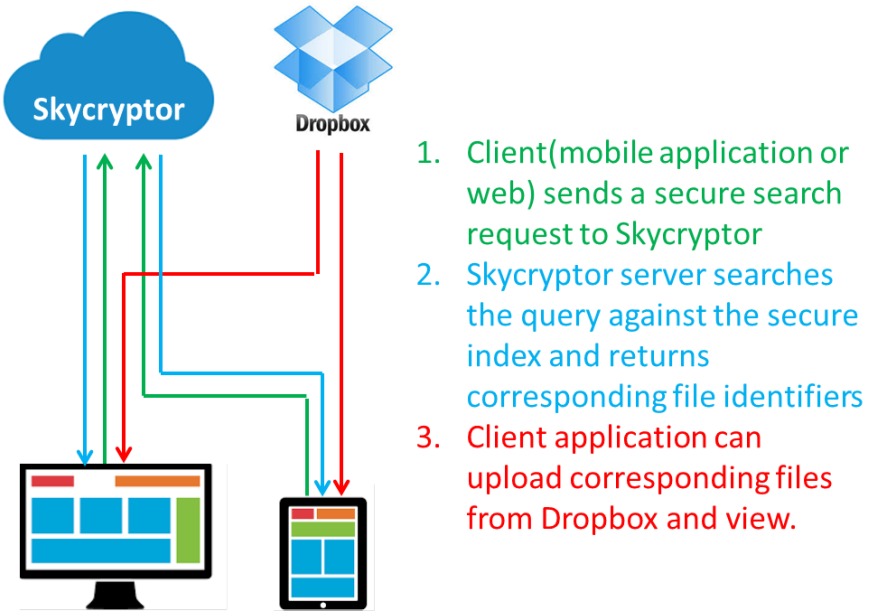

Fig: 3.1 (Secure Search Request)

3. The encrypted searchable encryption master key is stored in SkyCryptor servers.

4. When the user logins to web portal, his password-key is generated at client site in browser. When the user logins via client application, the password-key is generated at user device with the client application.
5. After authenticating successfully the user gets ciphertext of his searchable encryption key which can be decrypted with help of the password-key. All the other keys necessary for secure searching are then generated from the master key.
6. Having the searchable encryption keys recovered already the user can post search queries to the SkyCryptor servers. The search request flow is depicted in Figure 3.1.

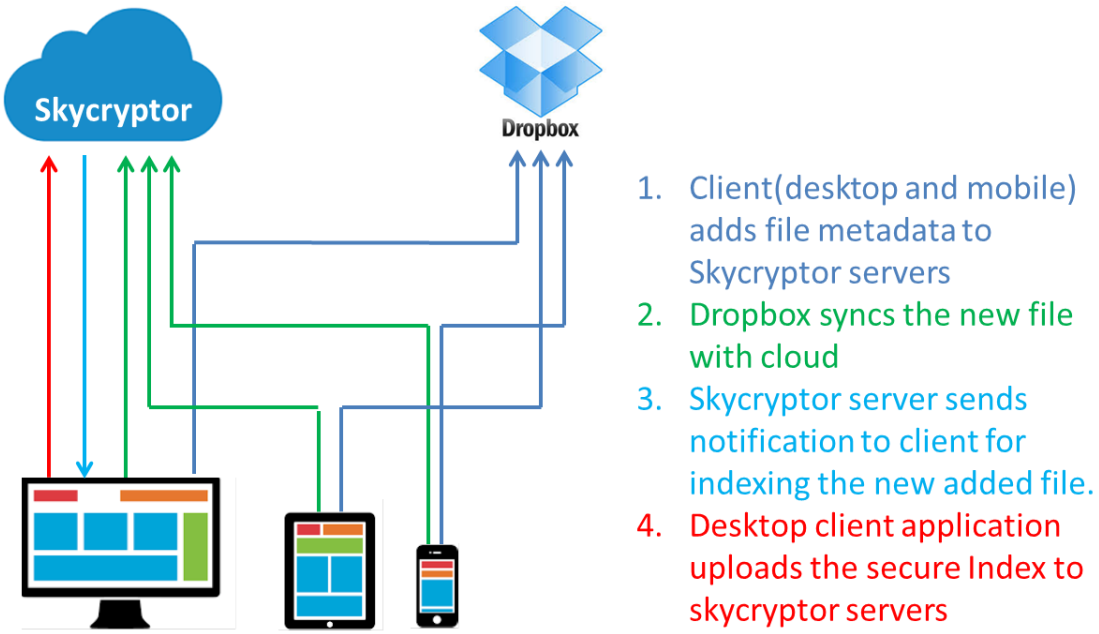

Fig: 3.2 (Secure Index Generation)

The SkyCryptor's client application, which is responsible for file encryption/decryption functionality, also allows to build a secure (encrypted) index on user files and which will be uploaded to SkyCryptor's server. The index generation flow is exposed in Figure 3.2 and is comprised of the following steps:

1. When the user encrypts a new file, the file metadata such as file name and path is posted to SkyCryptor server and the encrypted file is synced with the cloud service in parallel. Note that the user can add new files from both desktop and mobile devices.

2. The SkyCryptor server generates a new unique ID for each new added file and creates a special notification object which can prompt the user that the file should be indexed. The notification contains the file's local path as well as its unique ID.
3. The index file notifications can be processed only by the user desktop client applications. The desktop application time-by-time checks for new notifications and discovering a new notification starts processing it in the following way.
    a. The application decrypts and tokenizes the file content getting all unique words contained in the file
    b. The file unique ID contained in the notification acts as the file unique identifier in the secure index. The client application indexes the new added file according to the specified secure searchable indexing algorithm. Secure index is an encrypted index comprised of the encrypted keywords and encrypted file identifier in a specific form.
    c. The secure index is uploaded to SkyCryptor servers.

CHAPTER 4. AN EFFICIENT SPM PROTOCOL

In this chapter we introduce a SPM protocol. The initial version of this protocol for *regular expressions* with binary alphabet was published in proceedings of 10th international conference on computer science and information technologies (CSIT) Yerevan, Armenia after that the algorithm was extended to handle *regular expressions* with arbitrary length alphabets and was published in proceedings of NATO Advanced Research Workshop [**63**].

We consider such SPM problem where the first party (the client) has a *regular expression* r as a pattern[15] as long as the second party (the server) has a text string X which consists from the letters of the alphabet of the *regular expression*. Their objective is to check whether the string X fully matches the *regular expression* r or in other words if X belongs to the language generated by r ($X \in L(r)$) allowing both parties to learn the answer so that neither the client nor the server learned any additional information about the input of another. More formally the protocol calculates the function:

$$F_1(r, X) = \begin{cases} 1, \text{if } X \in L(r) \\ 0, \text{if } X \notin L(r) \end{cases}$$

Unlike other similar protocols [**43**], [**38**], [**36**], [**39**] our protocol uses no asymmetric cryptography operations and it is proved that it is private against a malicious one party (server) and fully secure against a malicious another party (client) (see the definitions in the section 4.1).

After all we provide modifications for the protocol to cover the following variation of the SPM problem:

a) The client or both parties want to learn whether the string X has a substring Y which matches the pattern r (partial matching):

$$F_2(r, X) = \begin{cases} 1, \text{if } \exists Y; Y \sqsubseteq X \text{ and } Y \in L(r) \\ 0, \text{otherwise} \end{cases}$$

Here $\sqsubseteq$ sign is used to denote that Y is a substring of X or equal to it.

[15] Check the definition of *regular expression* and the language it generates in section 1.2.

b) The client or both parties want to learn the number of substrings of the string X which match the pattern r (number of matches):

$$F_3(r, X) = |\{Y; Y \sqsubseteq X \; and \; Y \in L(r)\}|$$

Where $|\cdot|$ capacity (number of element) of the set defined inside $\{\cdot\}$ braces.

c) The client or both parties want to learn positions of all substrings of the string X which match the pattern r (positions of matches) :

$$F_4(r, X) = \{(i, j); 0 \leq i \leq j < |X| \; and \; X[i..j] \in L(r)\}$$

Where $|X|$ is the length of string X and $X[i..j]$ is the substring of X from i to j positions inclusive assumed that indexing of positions of X starts from 0.

Similarly to the construction by Mohassel P., Niksefat S., Sadeghian S. and Sadeghiyan B. described in "Efficient Protocol for Oblivious DFA Evaluation" [**43**] our protocol runs in a single client-server communication round and in both client and server sides it has the same as the protocol from [**43**] complexity of symmetric operations when the input alphabet of the *regular expression* r is binary. But unlike [**43**] our protocol has no restriction on the input alphabet of the *regular expressions* and it uses no asymmetric operations at all.

4.1 DEFINITIONS AND NOTATIONS

Deterministic finite automatons (DFA) [**105**] and *regular expressions* are equivalent formalisms **Invalid source specified.**, namely for each DFA it is possible to construct a *regular expression* which generates the same language as the DFA and vice versa, for each *regular expression* there is a DFA which accepts the same language as the *regular expression*. That is why in the description of the protocol we will assume that the client has a DFA Γ equivalent to the *regular expression* r, though in our C++ implementation of the protocol client's input is a *regular expression* which later gets converted to the equivalent DFA using corresponding conversion algorithm [**106**], [**107**].

We will use the following security definitions:

Secure Two-party Computation: Let $f = (f_1, f_2)$ of form $f: \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$ be a two party computation and π be a two-party protocol for computing f between

the parties p_1 and p_2. The input of p_1 is x and the input of p_2 is y. Here we will briefly define two notions of security.

a) Full security (simulation-based security) against malicious adversaries: This security level is defined by requiring indistinguishability (perfect, statistical or computational) between a real execution of the protocol and an ideal execution in which there is a TTP (trusted third party) who receives the parties input, evaluates the function and outputs the results to them.

b) Privacy against malicious adversaries: This level of security guarantees that a corrupted party will not learn any information about the honest parties input. However, this does not always guarantee that the joint outputs of the parties in the real world is simulatable in an ideal world.

For more detailed discussion of these security notations refer to [**108**].

Looking ahead we want to note that our protocol, we prove is full-secure against one malicious party and private against the other.

We will assume that the client has a DFA $\Gamma(Q; \Sigma; \Delta; s_1; F)$ with input alphabet $\Sigma = \{b_1, b_2, \dots, b_{|\Sigma|}\}$; set of states $Q = \{q_1, q_2, \dots, q_{|Q|}\}$; table of transitions $\Delta_{|Q|\times|\Sigma|}$, where $\Delta[q, b] \in Q$ is the state following after state q through letter b; initial state s_1 and set of accepting (or final) states F, while the server has an n length string $X \in \Sigma^n$. Saying the result of evaluation of Γ on the string X (or simply $\Gamma(X)$) we mean the Boolean value which is 1 (or $true$) if the state

$$\Delta\Big[\dots\Delta\Big[\Delta[s_1, X[1]], X[2]\Big]\dots, X[n]\Big]$$

is an accepting state (i.e. belongs to F) and 0 (or $false$) otherwise. By k we denote the security parameter of the protocol. The pipe-sign | is used to denote concatenation of strings and the circled plus sign $\oplus$ is used to denote per-bit XOR operation. By $\lceil r \rceil$ we denote the ceiling of a real number r. In matrix indexing sometimes we use letters of the input alphabet Σ and states Q instead of their numbers (i.e. b_j instead of j, q_j instead of j).

4.2 THE CASE OF DFA WITH BINARY ALPHABET

At first let us consider the case when the input alphabet of the DFA $\Sigma = \{0,1\}$ is binary, we call them binary DFAs.

Brief description of the protocol: The main steps of the protocol are the following:

a) *Client*: Create a special evaluation matrix (DFA matrix) M_Γ of size $n \times |Q| \times 2$ intended for evaluation of Γ on n-length binary strings.
b) *Client*: Create distorted DFA matrix GM_Γ by permuting each row of the M_Γ matrix, then encrypting each cell of the matrix using one time pad. As a result, to calculate $\Gamma(X)$ it will be enough for the server to have the distorted DFA matrix GM_Γ and a key for each position $i; 1 \leq i \leq n$ of the string X corresponding to the letter of that position.
c) *Server*: For each letter of string X create an OT query token [**94**] and send them all along with the OT initialization data to the client.
d) *Client*: For each OT query token create an OT response token for the corresponding key [**94**] and send them together with the distorted DFA matrix GM_Γ to the Server.
e) *Server*: From OT response tokens invoke the keys for positions of the string X, then compute $\Gamma(X)$ using those keys and GM_Γ distorted DFA matrix.

Now let us look at these steps in more detail.

Step a): For evaluating Γ on any n-length binary string we create a DFA matrix M_Γ of size $n \times |Q| \times 2$ such that for each state q and letter b $M_\Gamma[i, q, b] = \Delta[q, b]\ \forall i; 1 \leq i \leq n-1$ and $M_\Gamma[n, q, b] = 1$ if $\Delta[q, b]$ is a final state and $M_\Gamma[n, q, b] = 0$ if $\Delta[q, b]$ is not final. It is easy to see that in such notation $\Gamma(X)$ is equal to

$$M_\Gamma\left[n, \ldots M_\Gamma\left[2, M_\Gamma[1, s_1, X[1]], X[2]\right] \ldots, X[n]\right]$$

Fig: 4.1 (a, b) below illustrate an example of DFA and its DFA matrix.

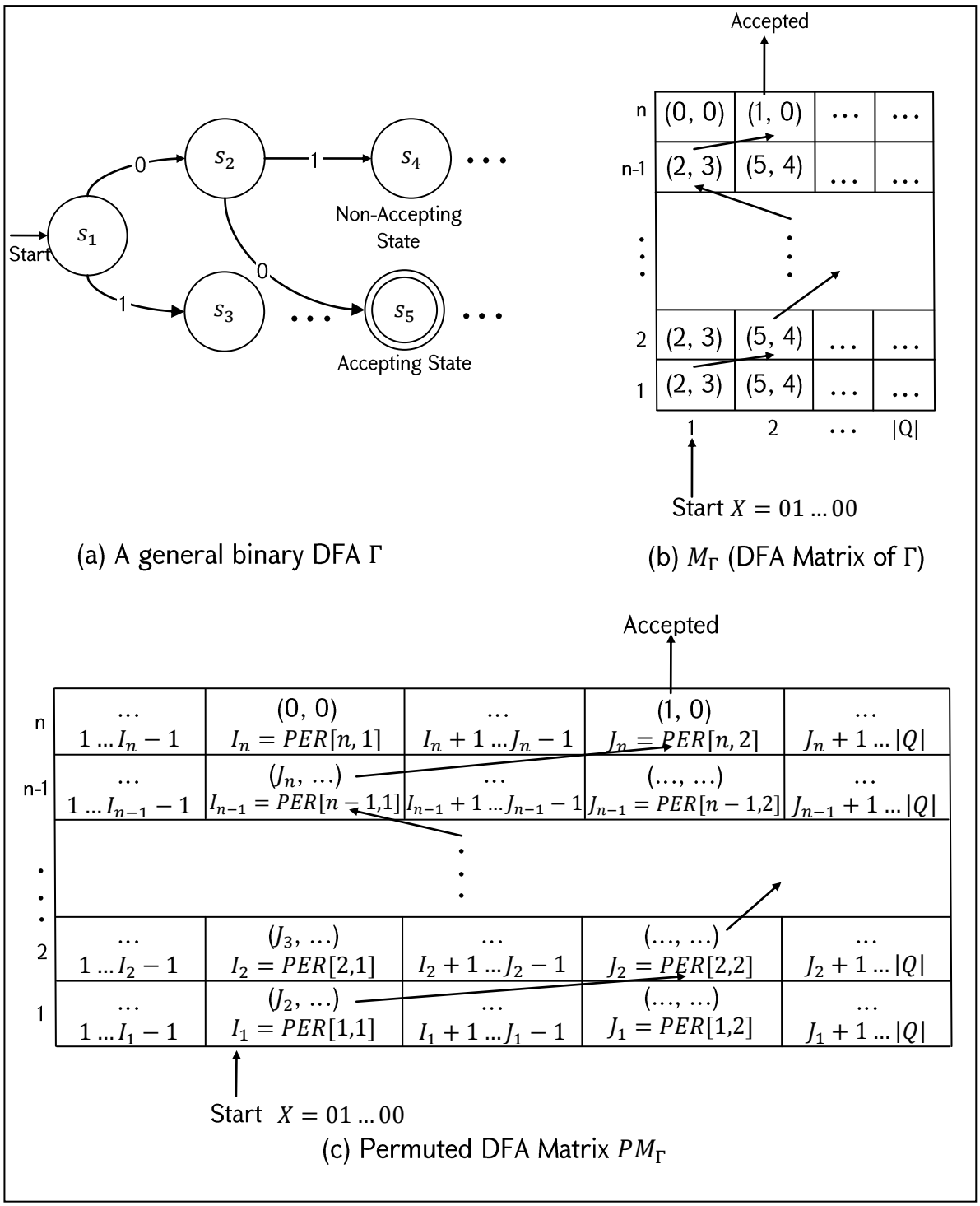

Fig: 4.1

Step b): Here it is worth to note that for any permutation $P: Q \to Q$ the DFA Γ is equal (i.e. accepts the same set of strings) to the DFA $\Gamma_P(Q; \{0,1\}; \Delta_P; P[s_1]; F_P)$ (where for each state q and letter $\Delta_P[P[q], b] = P[\Delta[q, b]]$; and $F_P = \{q: P^{-1}[q] \in F\}$). The first stage of DFA matrix distorting is based on this fact. Namely, we first create n random permutations of the

DFA states Q and fill by them n rows of a $n \times |Q|$ permutations matrix PER, then we create permuted DFA matrix PM_Γ such that for each state q and letter b

$$PM_\Gamma[i, PER[i,q], b] = PER\big[i+1, M_\Gamma[i,q,b]\big]$$

for each i; $1 \le i \le n-1$ and

$$PM_\Gamma[n, PER[n,q], b] = M_\Gamma[n,q,b]$$

Fig: 4.1 (c) above illustrates an example of a PM_Γ matrix. Here it is not hard to observe that in terms of PM_Γ matrix $\Gamma(X)$ is equal to

$$PM_\Gamma\Big[n, \dots PM_\Gamma\Big[2, PM_\Gamma\big[1, PER[1,s_1], X[1]\big], X[2]\Big] \dots, X[n]\Big]$$

For the second stage of DFA matrix distorting we generate an $n \times 2$ size matrix of random k-bit keys K and an $(n+1) \times |Q|$ size matrix of k'-bit pads PAD; where for each state q, and for each $i; 1 \le i \le n$ the $PAD[i,q]$ is a random k'-bit string and $PAD[n+1,q] = \underbrace{00\dots0}_{k'}$. Here by k' we denote $k - \lceil \log_2 |Q| \rceil$. For distorting the permuted DFA matrix we also need some CSPRNG (cryptographically secure pseudo-random number generator) $PRG: \{0,1\}^{k'} \to \{0,1\}^{2k}$. Since the PRG receives a k'-bit string as an input and returns $2k$-bit string as an output by $PRG(Y,0)$ we denote the first k-bits of $PRG(Y)$ and by $PRG(Y,1)$ we denote the second k-bits of $PRG(Y)$. Having all these, we create the distorted DFA matrix GM_Γ such that

$$GM_\Gamma[i,q,b] = \big(PM_\Gamma[i,q,b] \mid PAD\big[i+1, PM_\Gamma[i,q,b]\big]\big) \oplus K[i,b] \oplus PRG(PAD[i,q], b)$$

for each $i; 1 \le i \le n$, letter b and state q.

Step c): Having the distorted DFA matrix GM_Γ and $K\big[i, X[i]\big]$ for each $i; 1 \le i \le n$ the server will be able to calculate $\Gamma(X)$ (we will see that in step **e**). Hence, here for each letter $X[i]$ of its input string the server generates an OT query token $\mathrm{OT}_{Query}[i]$ and along with OT initialization info OT_{Init} sends them to the client. For the details of OT initialization info and OT query token generation see [**94**].

Step d): For each OT query the token $\mathrm{OT}_{Query}[i]$ $(1 \le i \le n)$ the client, using OT initialization info OT_{Init} generates an OT response token $\mathrm{OT}_{Responce}[i]$ which carries sufficient information to invoke $K\big[i, X[i]\big]$ (see [**94**]). Then the client sends those response tokens, distorted DFA matrix GM_Γ, $PER[1, s_1]$ and $PAD\big[1, PER[1, s_1]\big]$ to the server.

Step e): At this step first the server for each $i; 1 \leq i \leq n$ invokes $K\left[i, X[i]\right]$ key stored in the corresponding response token $\mathrm{OT}_{Responce}[i]$ (see [**94**]). Then having the keys, distorted DFA matrix GM_Γ, $PER[1, s_1]$ and $PAD\left[1, PER[1, s_1]\right]$,it runs the following algorithm to calculate $\Gamma(X)$:

Evaluation of distorted DFA matrix

$cur\_state\_id := PER[1, s_1];$

$cur\_pad := PAD[1, cur\_state\_id];$

for each row $i = 1$ to n **do**

$cur\_state\_id|cur\_pad := GM_\Gamma\left[i, cur_{state_{id}}, X[i]\right] \oplus K\left[i, X[i]\right] \oplus$

$PRG(cur\_pad, X[i]);$

end for

return cur_state_id

The step by step construction of the distorted DFA matrix implies that this algorithm will give the same result as evaluation of the initial DFA matrix, so no additional proof is required here.

These were all steps of the protocol for binary DFAs, now let us move to the case where there is no restriction on the input alphabet of the DFA.

4.3 CASE OF NON-BINARY DFA

Here again the protocol consists from 5 steps and they are almost the same as in the case of binary DFAs, the main difference is that the size of third dimension of M_Γ, PM_Γ and GM_Γ matrixes is $|\Sigma|$ instead of 2.

Let us look at these steps in detail.

Step a): Create a DFA matrix M_Γ of size $n \times |Q| \times |\Sigma|$ such that for each state q and letter $b \in \Sigma$ $M_\Gamma[i, q, b] = \Delta[q, b]$ $\forall i; 1 \leq i \leq n-1$ and $M_\Gamma[n, q, b]$ is equal to 1 if $\Delta[q, b]$ is a final state and it is 0 otherwise. In such notation $\Gamma(X)$ is equal to

$$M_\Gamma\Big[n, \dots M_\Gamma\Big[2, M_\Gamma\big[1, s_1, X[1]\big], X[2]\Big] \dots, X[n]\Big]$$

Fig: 4.2 (a, b) below illustrates an example of DFA with its DFA matrix.

Step b): Here again we create n random permutations of the DFA states Q and fill by them n rows of a $n \times |Q|$ permutations matrix PER, then we create permuted DFA matrix PM_Γ of size $n \times |Q| \times |\Sigma|$ such that for each state q and letter $b \in \Sigma$

$$PM_\Gamma[i, PER[i, q], b] = PER\big[i + 1, M_\Gamma[i, q, b]\big]$$

for each i; $1 \le i \le n - 1$ and

$$PM_\Gamma[n, PER[n, q], b] = M_\Gamma[n, q, b]$$

Fig: 4.2 (c) below illustrates an example of a PM_Γ matrix. In terms of PM_Γ matrix $\Gamma(X)$ is equal to

$$PM_\Gamma\Big[n, \dots PM_\Gamma\Big[2, PM_\Gamma\big[1, PER[1, s_1], X[1]\big], X[2]\Big] \dots, X[n]\Big]$$

Then we create an $n \times |\Sigma|$ size matrix of random k-bit keys K and an $(n + 1) \times |Q|$ size matrix of k'-bit pads PAD; here again for each state q, and for each $i; 1 \le i \le n$ the $PAD[i, q]$ is a random k'-bit string and $PAD[n + 1, q] = \underbrace{00 \dots 0}_{k'}$, where k' is $k - \lceil \log_2 |Q| \rceil$. We also need some CSPRNG $PRG: \{0,1\}^{k'} \to \{0,1\}^{k \cdot |\Sigma|}$, and by $PRG(Y,\ j)$ we denote the j-the k-bits of $PRG(Y)$ for each j; $1 \le j \le |\Sigma|$. After all, we create the distorted DFA matrix GM_Γ such that

$$GM_\Gamma[i, q, b] = \big(PM_\Gamma[i, q, b] \mid PAD\big[i + 1, PM_\Gamma[i, q, b]\big]\big) \oplus K[i, b] \oplus PRG(PAD[i, q], b)$$

for each $i; 1 \le i \le n$, letter $b \in \Sigma$ and state $q \in Q$.

Step c): And again having the distorted DFA matrix GM_Γ and $K\big[i, X[i]\big]$ for each $i; 1 \le i \le n$ the server will be able to calculate $\Gamma(X)$. So here we start OT phase between client and server for transferring corresponding keys and GM_Γ.

For each letter $X[i]$ of its input string the server generates an OT query token $\mathrm{OT}_{Query}[i]$ and along with OT initialization info OT_{Init} sends them to the client [**94**].

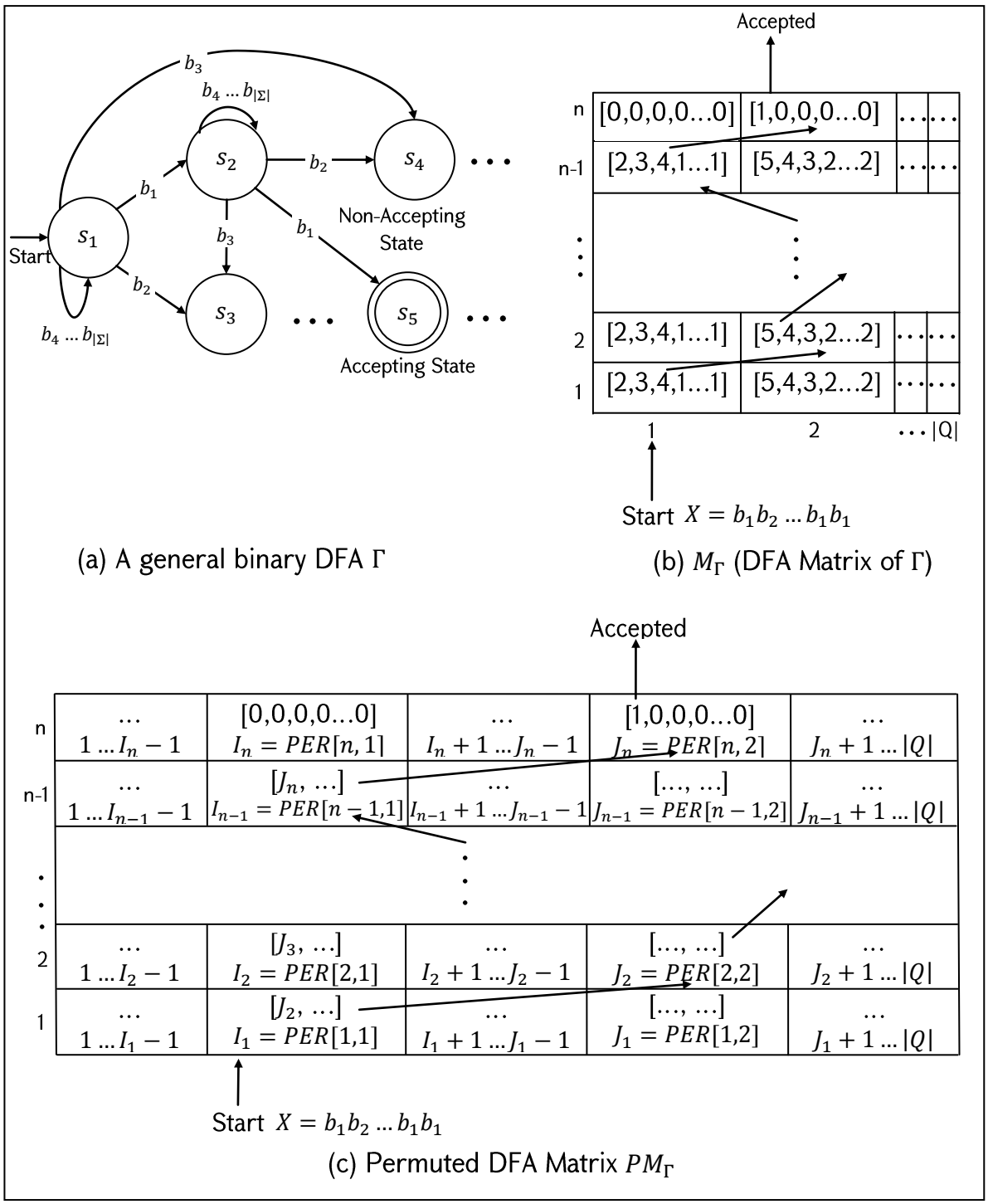

Fig: 4.2

Step d): For each OT query the token $\mathrm{OT}_{Query}[i]$ $(1 \le i \le n)$ the client, using OT initialization info OT_{Init} generates an OT response token $\mathrm{OT}_{Responce}[i]$ which carries sufficient info to invoke $K[i, X[i]]$ [**94**]. Then the client sends those response tokens, distorted DFA matrix GM_Γ, $PER[1, s_1]$ and $PAD[1, PER[1, s_1]]$ to the server.

Step e): At this step firstly the server invokes $K[i, X[i]]$ keys stored in the corresponding response token $\mathrm{OT}_{Responce}[i]$ for each $i; 1 \leq i \leq n$ [**94**]. Then having the keys, distorted DFA matrix GM_{Γ}, $PER[1, s_1]$ and $PAD[1, PER[1, s_1]]$,it runs the following algorithm to calculate $\Gamma(X)$:

Evaluation of distorted DFA matrix

$cur\_state\_id := PER[1, s_1];$

$cur\_pad := PAD[1, cur\_state\_id];$

for each row $i = 1$ to n **do**

$cur\_state\_id|cur\_pad := GM_{\Gamma}\left[i, cur_{state_{id}}, X[i]\right] \oplus K[i, X[i]] \oplus$

$PRG(cur\_pad, X[i]);$

end for

return cur_state_id

Here again the step by step construction of the distorted DFA matrix implies that this algorithm will give the same result as evaluation of the initial DFA matrix.

4.4 PROTOCOL SECURITY

In our construction we use a WB based 1-out-of-n OT protocol. Such *OT protocol is considered to be secure* if the underlying WB encryption schema is secure. In our case we use WB encryption schema based on SAFER+ encryption schema. WB scheme can be considered secure if no computationally bounded adversary is able to extract the master encryption key from the WB encryption tables and no computationally bounded adversary is able to make decryption functionality with help of only the WB encryption tables. So WB scheme is considered to be secure if it is secure against key-recovery and reverse-engineering attacks.

Taking into account this and security definitions from section 4.1 we see that according to the theorem 1 from [**43**] this protocol is *fully-secure when only one party is malicious* and it is *private when both parties are malicious*.

4.5 IMPLEMENTATION

A C++ library has been created which implements the protocol. The implementation is parameterized by the security parameter k. The main class is Gdfa template (Fig: 4.3 and 4.4).

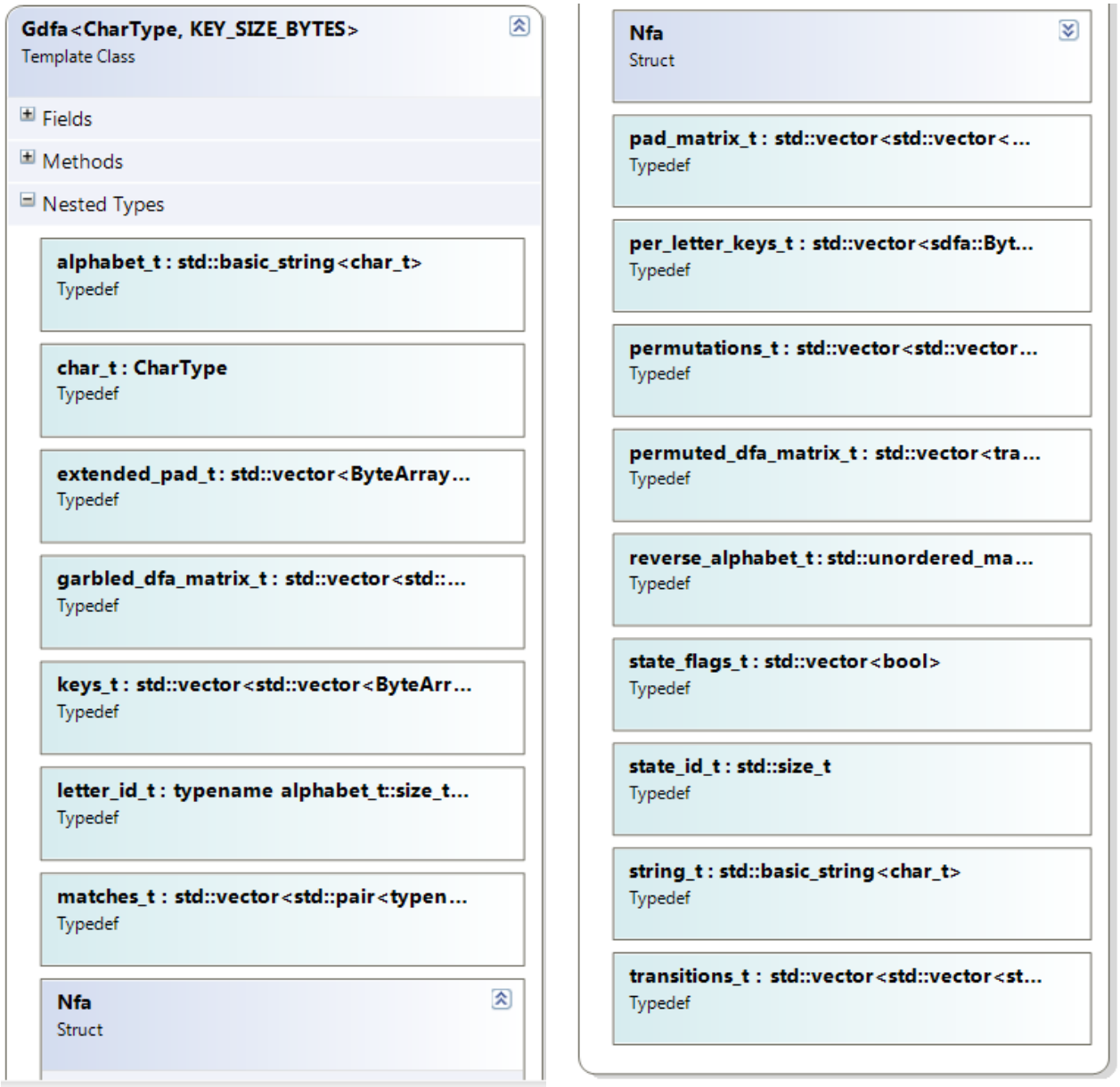

Fig: 4.3

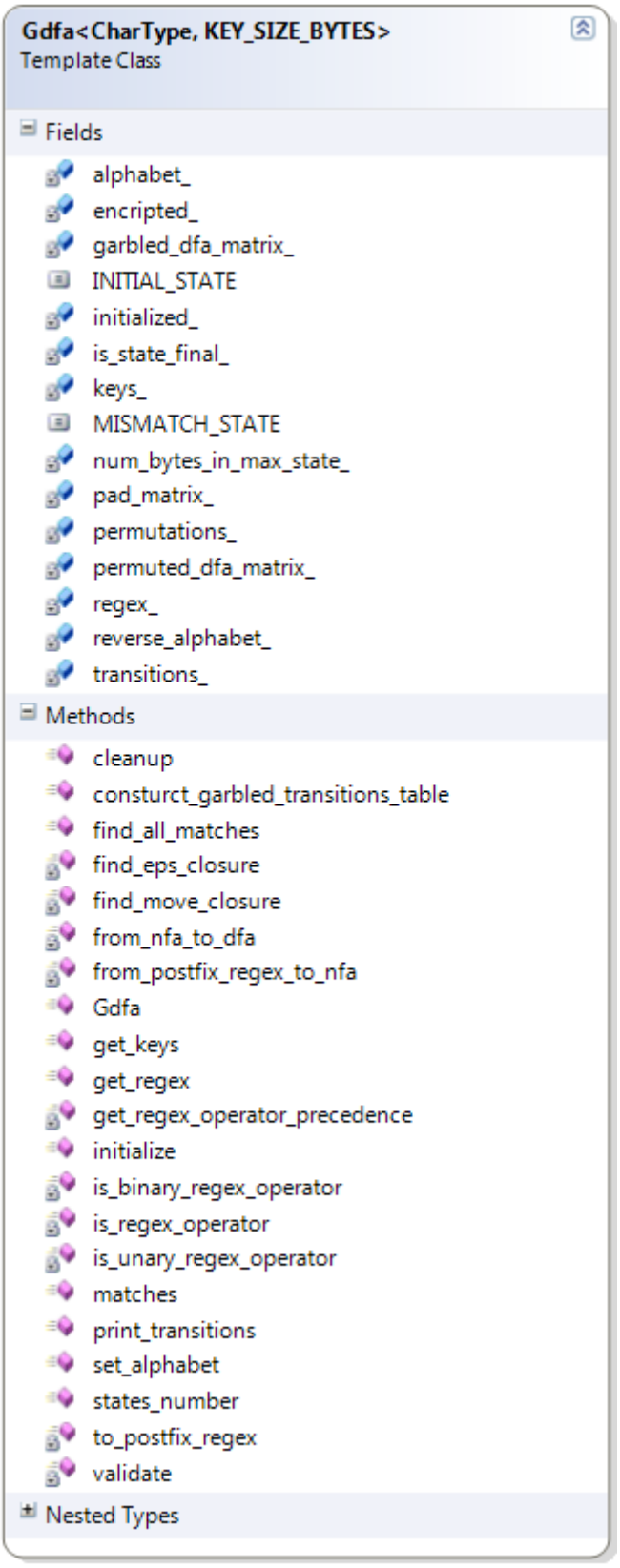
Gdfa<CharType, KEY_SIZE_BYTES>
Template Class
Fields
alphabet_
encripted_
garbled_dfa_matrix_
INITIAL_STATE
initialized_
is_state_final_
keys_
MISMATCH_STATE
num_bytes_in_max_state_
pad_matrix_
permutations_
permuted_dfa_matrix_
regex_
reverse_alphabet_
transitions_
Methods
cleanup
consturct_garbled_transitions_table
find_all_matches
find_eps_closure
find_move_closure
from_nfa_to_dfa
from_postfix_regex_to_nfa
Gdfa
get_keys
get_regex
get_regex_operator_precedence
initialize
is_binary_regex_operator
is_regex_operator
is_unary_regex_operator
matches
print_transitions
set_alphabet
states_number
to_postfix_regex
validate
Nested Types

Fig: 4.4

As a client API class Gdfa allows to create the distorted DFA matrix from the input regex and as a server API class it has find_all_matches() method which returns all positions of substrings which match the distorted DFA.

The steps of the protocol are implemented in the demo application. The application acts as both a client and a server. As a server, it receives a text string as an input, and as a client, its input is a *regular expression* which it converts to the equivalent DFA using Thompson's construction algorithm [**106**] to create NDFA equivalent to the *regular expression* then using subset construction algorithm [**107**] to convert the NDFA to the DFA. After all it runs the steps a) – e) of the protocol and calculates the result of evaluation of the DFA on the text string. During calculations it prints out time spent for different parts of computations.

4.6 PERFORMANCE TESTING

Algorithm construction implies that overall number of computations depends on the security parameter k, the number of the DFA states $|Q|$, the length of the text string n and the number of letters in the DFAs input alphabet $|\Sigma|$. Besides that, in the algorithm we have generation of random pads and keys, as well as usage of CSPRNG, and depending on the approach used for generation of random pads and keys and chosen CSPRNG the efficiency of the algorithm may differ for the same input data (k, $|Q|$, $|\Sigma|$ and $|X|$). We did our benchmarks for $k = 128$ and $|\Sigma| = 2$ on a 64-bit Windows 7 PC with Intel® Core™ 2 Quad Q6600 2.4 GHz processor and 4GB RAM, using SHA-256 as $\{0,1\}^{k'} \to \{0,1\}^{k\cdot|\Sigma|}$ CSPRNG and C++ standard library's rand() function for random pad/key generation. The method of distorted DFA matrix construction shows and our benchmarks confirmed that the number of operations, hence the spent time for distorted DFA matrix creation, is proportional to the $|Q| \cdot |\Sigma| \cdot n$ multiplication. The first table shows performance of distorted DFA matrix generation of our implementation on inputs of different sizes. The table contains averaged results from 10 runs for each pair of inputs.

Table 4.1. Distorted DFA creation time when (sec):

$\lvert Q\rvert$ \ n	10	100	1000	10000	100000
10	<0.001	0.002	0.017	0.17	1.7
100	0.002	0.017	0.17	1.7	17
1000	0.017	0.17	1.7	17	>100
10000	0.17	1.7	17	>100	>100
100000	1.7	17	>100	>100	>100
1000000	17	>100	>100	>100	>100

The second table shows the performance of the OT phase of the protocol and performance of the distorted DFA matrix evaluation by the server.

Table 4.2. Time spent in OT phase when (sec):

n	OT query generation and response extraction (server)	OT response generation (client)	Distorted DFA evaluation (server)
10	<0.001	<0.001	<0.001
100	0.002	<0.001	<0.001
1000	0.021	0.007	<0.001
10000	0.21	0.07	0.002
100000	2.1	0.7	0.02
1000000	21	7	0.2

The table shows only the dependency of efficiencies of the operations from n since the number of OT queries/responses and the number of steps for distorted DFA matrix evaluation is equal to n and do not depend on structure of DFA.

4.7 CONCLUSION

Unlike other protocols for oblivious DFA evaluation published in recent years [**31**], our protocol is totally free from public-key operations, and it makes it somewhat unique among others. Table 4.3 below illustrates complexities of client and server computations and network communication bandwidth of our and other recent protocols.

Table 4.3. Operation complexities of protocols

	Round Complexity	Client Computations		Server Computations		Network Bandwidth
		Asymmetric	Symmetric	Asymmetric	Symmetric	
Troncoso [**36**]	$O(n)$	$O(n\|Q\|)$	None	$O(n\|Q\|)$	$O(n\|Q\|)$	$O(n\|Q\|k)$
Frikken [**38**]	2	$O(n+\|Q\|)$	$O(n\|Q\|)$	$O(n+\|Q\|)$	$O(n\|Q\|)$	$O(n\|Q\|k)$
Gennaro [**39**]	$O(\min\{\|Q\|,n\})$	$O(n\|Q\|)$	None	$O(n\|Q\|)$	None	$O(n\|Q\|k)$
Yao [**28**]	1	$O(n)$	$O(n\|Q\|\log\|Q\|)$	$O(n)$	$O(n\|Q\|\log\|Q\|)$	$O(kn^2)$
Ishai [**86**]	1	$O(n)$	None	$O(n\|Q\|)$	None	$O(n\|Q\|k)$
Mohassel [**43**]	1	$O(n)$	$O(n)$	$O(n)$	$O(n\|Q\|)$	$O(n\|Q\|k)$
This Protocol $\|\Sigma\|=2$	1	None	$O(n\|Q\|)$	None	$O(n)$	$O(n\|Q\|k)$

It is also worth to note that our protocol allows both parties to learn only $\Gamma(X)$, but in some applications it may be inconvenient or insufficient. In [**43**] it is shown that for each of the following modifications of the problem it is possible to solve it after modifying their protocol a little, and that those modifications have no security leakage. All those modification of their protocol work for our protocol as well.

a. The client wants to hide the answer from the server.
b. The client or both parties want to learn whether the string X has a substring Y such that Γ recognizes Y.
c. The client or both parties want to learn positions of all substrings of the string X recognized by Γ.

d. The client or both parties want to learn the number of substrings of the string X recognized by Γ.

CONCLUSION

At the first time the problem of construction of a searchable encryption (SE) scheme was explicitly considered in 2000 by Dawn Xiaodong Song, David Wagner and Adrian Perrig in their paper “Practical Techniques for Search on Encrypted Data”. Since that various SE schemes have been propose which differ by:

1. The provided level of security and privacy;
2. The intended number of users;
3. The dynamicity (either supports data update queries or not);
4. Performance, memory usage, transferred data and client-server communication rounds.

These schemes represented as set of protocols for deploying of a set of files $\boldsymbol{f} = \{f_1, f_2, \dots, f_n\}$ to a remote storage and working with these files after that. The protocols are mainly of three types:

1. *Initialization protocol*: in this stage based on the security parameter k of the scheme and the initial set of files $\boldsymbol{f}$ keys $\boldsymbol{K}$ of the scheme, initial secure index $\boldsymbol{I}$ and ciphertexts $\boldsymbol{c}$ of the files $\boldsymbol{f}$ are generated and the pair $(\boldsymbol{I}, \boldsymbol{c})$ is stored to the remote server.
2. *Modification protocols* (for file addition, deletion, edit): in this stage the corresponding update token (or tokens) is generated by the user and passed to the server which updates the pair $(\boldsymbol{I}, \boldsymbol{c})$ according to the instruction (add, delete, modify) carried by the token(s). In some update protocols tokes are not passed at once, but the processing of the instruction is done during multiple communication rounds.
3. *Search protocols*: in this stage the corresponding search token (or tokens) which hide information about one or more keywords is generated by the user and passed to the server. The later one using this token(s) finds the set of file ids associated with the hidden keywords and passes them (probably also the set of ciphertexts of these files) to the user.

In this dissertation we provided a survey of the most part of SE schemes published in recent years, and discussed their pros and cons based on the characteristics form the first list [**61**]. Also we introduced a novel method for delegating the search functionality to third party users in symmetric searchable encryption schemes and provided its implementation.

The knowledge obtained during the research on SE schemes was served as a basis in launching a startup project SkyCryptor by Aram Jivanyan and team [**59**]. SkyCryptor intended to provide security services for the users of public cloud storages (Dropbox and Google Drive for the beginning). In particular it uses a comprehensive searchable encryption scheme to provide both search and sharing functionals. And within the SkyCryptor project another more practical problems have been considered, which we provide as a part of this dissertation.

Within the research of SkyCryptor project we reviewed functionality of provided services by the other players of the industry (Sookasa [**55**], boxcryptor [**56**], nCryptedCloud [**54**], etc.) and understood the aspects of implementation of zero-knowledge search functionality in encrypted cloud storages [**60**]. As a result of our research a zero-knowledge architecture for supporting queries over encrypted cloud file have been designed an integrated into the SkyCryptor.

The next problem considered is this thesis is the special case of multi-party secure function evaluation problem denoted as “secure pattern matching/search” problem. After the first publication in 1982 where secure two party computations were considered for the first time by Andrew Chi-Chih Yao in the paper “Protocols for secure computations” [**28**] the problem of multi-party secure function evaluation become a topic of active research. Besides that its special case the two-party SPM problem, also become a hot investigation topic as a part of the more generic problem as well as a standalone problem.

In this work we provided our protocol for oblivious *regular expression* matching (or DFA evaluation) [**63**]. It is similar to the protocol described in the paper “An Efficient Protocol for Oblivious DFA Evaluation and Applications” by Mohassel P., Niksefat S., Sadeghian S. and Sadeghiyan B. [**43**] and like that protocol provides full security against

malicious client and privacy against malicious server. The advantage of our protocol is that it does not use any public key operation in its algorithms and supports *regular expressions* with arbitrary length input alphabets. In the dissertation benchmarks of the implementation are provided showing its high performance.

REFERENCES

[1] Rajesh S., Swapna S., Reddy P. S., "Data as a Service (Daas) in Cloud Computing," *Global Journal of Computer Science and Technology: Cloud & Distributed*, vol. 12, no. 11, 2012.

[2] Hacigümüs H., Iyer B., Mehrotra S., "Providing Database as a Service," *In Proc. of 18th International Conference on Data Engineering (ICDE)*, 2002.

[3] Software & Infromation Industry Association (www.siia.net), *Software as a Service: Strategic Backgrounder*. Washington D. C., 2001.

[4] (2006) Amazon S3, Cloud Computing Storage for Files, Images, Videos. [Online]. https://aws.amazon.com/s3/

[5] (2010) Microsoft Azure Storage. [Online]. https://azure.microsoft.com/services/storage

[6] (2007) Dropbox: file hosting service. [Online]. https://www.dropbox.com/

[7] (2012) Google Drive: Shared disk. [Online]. https://drive.google.com/

[8] (2005) Box | Secure Content & Online File Sharing for Businesses. [Online]. https://www.box.com/

[9] (2007) Microsoft OneDrive: file hosting service. [Online]. https://onedrive.live.com/

[10] Mather T., Kumaraswamy S., Latif S., *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance.*: O'Reilly Media, Inc., 2009.

[11] Neuman B. C., Ts'o T., "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 33-38, 1994.

[12] Arkills B., *LDAP Directories Explained: An Introduction and Analysis.*: Addison-Wesley Professional, 2003.

[13] Song D. X., Wanger D., Perrig A., "Practical Techniques for Search on Encrypted Data," *In Proceedings of IEEE Symposium on Security and Privacy*, 2000.

[14] Goh E.-J., "Secure indexes," *Cryptology ePrint Archive*, no. 216, 2003. [Online]. http://eprint.iacr.org/2003/216/

[15] Hacigümüs H., "Privacy in Database-as-a-Service Model," University of California, PhD Thesis 2003.

[16] Bellare M., Boldyreva A., O'Neill A., "Deterministic and Efficiently Searchable Encryption," *In Proc. of the 27th Annual International Cryptology Conference on Advances in Cryptology*, pp. 535-552, 2007.

[17] Kamara S., Papamanthou C., Roeder T., "Dynamic Searchable Symmetric Encryption," *In Computer and Communications (CCS)*, pp. 965–976, 2012.

[18] Premasathian N., "Searchable Encryption Schemes: With Multiplication and Simultaneous Congruences," *International Conference on Information Security and Cryptology (ISCISC)*, 2012.

[19] Van Liesdonk P., Sedghi S., Doumen J., Hartel P., Jonker W., "Computationally Efficient Symmetric Encryption," *In Proc. of Secure Data Management (SDM) 7th VLDB Workshop*, 2010.

[20] Kamara S., Papamanthou C., "Parallel and Dynamic Searchable Symmetric Encryption," *Financial cryptography and data security*, pp. 258–274, 2013.

[21] Chang Y.-C., Mitzenmacher M., "Privacy Preserving Keyword Searches on Remote Encrypted Data," *Lecture Notes in Computer Science: Applied Cryptography and Network Security*, vol. 3531, pp. 442-455, 2005.

[22] Park D. J., Cha J., Lee P. J., "Searchable Keyword-Based Encryption," *Cryptology ePrint Archive*, no. 367, 2005.

[23] Chase M., Kamara S., "Structured Encryption and Controlled Disclosure," *Advances in Cryptology - ASIACRYPT*, vol. 5477, pp. 577-594, 2010.

[24] Chai Qi, Gong G., "Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers," *IEEE International Conference on Communications (ICC)*, pp. 917-922, 2012.

[25] Devlin K. J., *Mathematics: The Science of Patterns: The Search for Order in Life, Mind and the Universe*, 1, Ed.: Scientific American Library, 1997.

[26] Lawson M. V., *Finite Automata.*: CRC Press, 2003, pp. 98-100.

[27] Sheng Yu, *Handbook of Formal Languages: Volume 1. Word, Language, Grammar.*: Springer, 1997, pp. 41-110.

[28] Yao A. C., "Protocols for secure computations," *In proc. of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 160-164, 1982.

[29] Chaum D., Crépeau C., Damgard I., "Multiparty Unconditionally Secure Protocols," *In Proc. of the 20th Annual ACM Symposium on Theory of Computing*, pp. 11-19, 1988.

[30] Lindell Y., Pinkas B., "An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries," *EUROCRYPT '07 In Proc. of the 26th Annual International Conference on Advances in Cryptology*, pp. 52-78, 2007.

[31] Koleshnikov V., Sadeghi A.-R., Schneider T., "From Dust to Dawn: Practically Efficient Two-Party Secure Function Evaluation Protocols and their Modular Design," *IACR Cryptology ePrint Archive*, no. 79, 2010.

[32] Koleshnikov V., Sadeghi A.-R., Schneider T., "A Systematic Approach to Practically Efficient General Two-Party Secure Function Evaluation Protocols and Their Modular Design," *Journal of Computer Security archive*, vol. 21, no. 2, pp. 283-315, 2013.

[33] Shelat A., Shen C., "Fast Two-Party Secure Computation with Minimal Assumptions," *In Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CSS '13)*, 2013.

[34] Frederiksen T. K., Nielsen J. B., "Fast and Maliciously Secure Two-Party Computation Using the GPU," *Applied Cryptography and Network Security: Lecture Notes in Computer Science*, vol. 7954, pp. 339-356, 2013.

[35] Huang Y., Katz J., Evans D., "Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose," *Advances in Cryptology: Lecture Notes in Computer Science*, vol. 8043, pp. 18-35, 2013.

[36] Troncoso-Pastoriza J. R., Katzenbeisser S., Celik M., "Privacy preserving error resilient dna searching," *In Proc. of the 14th ACM Conference on Computer and*

Communications, pp. 519-528, 2007.

[37] Hazay C., Lindell Y., "Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries," *Theory of Cryptography*, vol. 4948, pp. 155-175, 2008.

[38] Frikken K. B., "Practical Private DNA String Searching and Matching through Efficient Oblivious Automata Evaluation," *Data and Applications Security XXIII*, vol. 5645, pp. 81-94, 2009.

[39] Gennaro R., Hazay C., Sorensen J. S., "Text Search Protocols with Simulation Based Security," *Lecture Notes in Computer Science: Public Key Cryptography – PKC*, vol. 6056, pp. 332-350, 2010.

[40] Hazay C., Toft T., "Computationally Secure Pattern Matching in the Presence of Malicious Adversaries," *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, pp. 195-212, 2010.

[41] Vergnaud D., "Efficient and Secure Generalized Pattern Matching via Fast Fourier Transform," *Progress in Cryptology – AFRICACRYPT*, vol. 6737, pp. 41-58, 2011.

[42] Baron J., Defrawy E. K., Minkovich K., Ostrovsky R., Tressler E., "5PM: secure pattern matching," *SCN'12 Proceedings of the 8th international conference on Security and Cryptography for Networks*, pp. 222-240, 2012.

[43] Mohassel P., Niksefat S., Sadeghian S., Sadeghiyan B., "An Efficient Protocol for Oblivious DFA Evaluation and Applications," *In Proceedings of Cryptographers' Track at the RSA Conference*, pp. 398-415, 2012.

[44] El Defrawy K., Faber S., "Blindfolded Data Search via Secure Pattern Matching," *Computer*, vol. 46, no. 12, pp. 68-75, 2013.

[45] Baldi P., Baronio R., De Cristofaro E., Gasti Paolo, Tsudik G., "Countering Gattaca: Efficient and Secure Testing of Fully-Sequenced Human Genomes," *In Proc. of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, 2011.

[46] (2002, December) Privacy with Security. Technical report, DARPA Information Science and Technology Study Group. [Online]. http://www.cs.berkeley.edu/~tygar/papers/

[47] Kurosawa K., Ohtaki Y., "UC-Secure Searchable Symmetric Encryption," *Financial Cryptography and Data Security: Lecture Notes in Computer Science*, vol. 7397, pp. 285-298, 2012.

[48] Kurosawa K., Ohtaki Y., "How to Construct UC-Secure Searchable Symmetric Encryption Scheme," *Cryptology ePrint Archive*, no. 251, 2015.

[49] Stefanov E., Papamanthou C., Shi E., "Practical Dynamic Searchable Encryption with Small Leakage," *NDSS Symposium*, 2014.

[50] Naveed M., Prabhakaran M. Gunter C. A., "Dynamic Searchable Encryption via Blind Storage," *Security and Privacy (SP), IEEE Symposium*, pp. 639-654, 2014.

[51] Strizhov M., Ray I., "Multi-Keyword Similarity Search Over Encrypted," *Cryptology ePrint Archive*, no. 137, 2015.

[52] Zhao F., Nishide T., Sakurai K., "Multi-user keyword search scheme for secure data sharing with fine-grained access control," *In ICISC*, pp. 406-418, 2011.

[53] Yang Y., Lu H., Weng J., "Multi-user private keyword search for cloud computing," *In CloudCom*, pp. 264-271, 2011.

[54] (2012) nCryptedCloud | Enterprise Grade Secure Cloud File Sharing and Collaboaration. [Online]. https://www.encryptedcloud.com/

[55] (2012) Sookasa | Fully integrated CASB and cloud security provider. [Online]. https://www.sookasa.com/

[56] (2011) Encryption Software to Secure Files in the Cloud | Boxcryptor. [Online]. https://www.boxcryptor.com

[57] (2014) Mylar: Platform for building secure web applications. [Online]. https://css.csail.mit.edu/mylar/

[58] Popa R. A., Zeldovich N., "Multi-Key Searchable Encryption," *Cryptology ePrint Archive*, no. 508, 2013.

[59] (2016) SkyCryptor: Cloud Securing Gateway. [Online]. https://www.skycryptor.com/

[60] Jivanyan A., Hovsepyan M., "Implementation Aspects of Search Functionality Over Encrypted Cloud Data," *Transactions of IIAP of the NAS RA, Mathematical Problems of Computer Science*, vol. 44, pp. 101-108, Yerevan, Armenia, 2015.

[61] Hovsepyan M., "Review of Searchable Encryption Algorithms," *"Вестник" Российско-Армянский Университет*, vol. 2, pp. 39-53, Ереван, Армения, 2015.

[62] Khachatryan G., Hovsepyan M., Jivanyan A, "Efficient Secure Pattern Search Algorithm," *In Proc. of 10th International Conference on Computer Science and Information Technologies (CSIT) and IEEE: Conference*, pp. 147-151 in CSIT and 90-94 in IEEE, Yerevan, Armenia, 2015.

[63] Khachatryan G., Hovsepyan M., Jivanyan A., "Two-Party Regular Expression Matching Protocol without Asymmetric Cryptography Operations," *Transactions of IIAP of the NAS RA, Mathematical Problems of Computer Science*, vol. 45, pp. 77-89, Yerevan, Armenia, 2016.

[64] Even S., Goldreich O., Lempel A., "A Randomized Protocol for Signing Contracts," *Communications of the ACM*, vol. 28, no. 6, pp. 637-647, 1985.

[65] Curtmola R., Garay J., Kamara S., Ostrovsky R., "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," *Journal of Computer Security*, pp. 79-88, 2011.

[66] Cash D., Jarecki S., Jutla C. S., Krawczyk H., Rosu M.-C., Steiner M., "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries," *In R. Canetti and J. A. Garay, editors, CRYPTO Part I, volume 8042 of LNCS*, pp. 353–373, 2013.

[67] Kallahalla M., Riedel E., Swaminathah R., Wang Q., Fu K. E., "Plutus: Scalable Secure File Sharing on Untrusted Storage," *In Proc. of the 2nd USENIX Conference on File and Storage Technologies*, pp. 29-42, 2003.

[68] Boneh D., Waters B., "Conjunctive, subset, and range queries on encrypted data," *In TCC*, pp. 535-554, 2007.

[69] Jarecki S., Jutla C., Krawczyk H., Rosu M. C., Steiner M., "Outsourced Symmetric Private Information Retrieval," *In proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[70] Chor B., Kushilevitz E., Goldreich O., Sudan M., "Private Information Retrieval," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 965-981, 1998.

[71] Yekhanin S., "Private Information Retrieval," *Communications of the ACM*, vol. 53, no. 4, pp. 68-73, 2010.

[72] Noar M., Pinkas B., "Oblivious Transfer and Polynomial Evaluation," *In Proc. of the 31st Annual ACM Symposium on Theory of Computing (SOTC)*, pp. 245-254, 1999.

[73] Asharov G., Lindell Y., Schneider T., Zohner M., "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation," *In Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pp. 535-548, 2013.

[74] Rabin M. O., "How to exchange secrets by oblivious transfer," Aiken Computation Laboratory, Harvard University, Technical Report TR-81 1981.

[75] Kilian J., "Founding Cryptography on Oblivious Transfer," *In Proc. of the 20th Annual ACM Symposium on the Theory of Computation (STOC)*, pp. 20-31, 1988.

[76] Bellare M., Rogaway P., "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols," *In proc. of the ACM Conference on Computer and Communications Security (CCS)*, 1993.

[77] Goldreich O., Ostrovsky R., "Software Protection and Simulation on Oblivious RAMs," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431-473 , 1996.

[78] Kushilevitz E., Lu S., Ostrovsky R., "On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme," *In Proc. of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 2011.

[79] Stefanov E., Van Dijk M., Shi E., Fletcher C., Ren L., Yu X., Devadas S., "Path ORAM: an extremely simple oblivious RAM protocol," *In Proc. of the ACM SIGSAC Conference on Computer & Communications Security*, pp. 299-310 , 2013.

[80] Bloom B. H., "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM (CACM)*, vol. 13, no. 7, 1970.

[81] Bellovin S. M., Cheswick W. R., "Privacy-Enhanced Searches Using Encrypted Bloom Filters," *Cryptology ePrint Archive*, 2004 https://eprint.iacr.org/2004/022.pdf.

[82] Cash D., Jaeger J., Jarecki S., Jutla C., Krawczyk H., Rosu M.C., Steiner M., "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation," *Cryptology ePrint Archive*, no. 853, 2014.

[83] Boneh B., Di Crescenzo G., Ostrovsky R., Persiano G., "Public Key Encryption with Keyword Search," *Lecture Notes in Computer Science: Advances in Cryptology - EUROCRYPT*, vol. 3027, pp. 506-522, 2004.

[84] Croft W. B., Metzler D., Strohman T., *Search Engines: Information Retrieval in Practice*. USA: Pearson, 2010.

[85] Katz J., Malka L., "Secure Text Processing with Applications to Private DNA Matching," *In Proc. of of the 17th ACM Conference on Computer and Communications Security (CCS)*, pp. 485-492, 2010.

[86] Ishai Y., Paskin A., "Evaluating Branching Programs on Encrypted Data," *Theory of Cryptography*, vol. 4392, pp. 575-594, 2007.

[87] Ebbinghaus, H.-D., Flum, J., Thomas W., *Mathematical Logic.*: Springer, 1994, p. 656.

[88] Massey J., Khachatrian G., Kuregian M., "Nomination of SAFER+ as Candidate Algorithm for the Advanced Encryption Standard (AES)," 1998.

[89] Chow S., Eisen P., Johnson H., van Oorschot. P.C., "White-Box Cryptography and an AES Implementation," *In 9th Annual Workshop on Selected Areas in Cryptography (SAC)*, pp. 15-16, 2002.

[90] Khachatryan G., Kuregyan M., Abrahamyan S., Jivanyan A., Karapetyan M, Oleynik A., "Design and Cryptanalysis of Secure White-Box Encryption Based on SAFER+ Algorithm," *(submitted for publication)*, see more in http://cse.aua.am/applied-cryptography-laboratory/.

[91] Lipmaa H., "An Oblivious Transfer Protocol with Log-Squared Communication.," *In Proc. of the 8th Information Security Conference (ISC)*, vol. 3650, pp. 314-328, 2005.

[92] Noar M. Pinkas B., "Efficient Oblivious Transfer Protocols," *In Proc. of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 448-457, 2001.

[93] Peikert C., Vaikuntanathan V., Waters B., "A Framework for Efficient and Composable Oblivious Transfer," *Advances in Cryptology (CRYPTO)*, pp. 554-571, 2008.

[94] Khachatryan G., Jivanyan A., "Efficient Oblivious Transfer Protocols based on White-Box Cryptography," *(submitted for publication)*, see more in http://cse.aua.am/applied-cryptography-laboratory/.

[95] Rivest R., Smith A. C., Fu K. E., "Group Sharing and Random Access in Cryptographic Storage File Systems," MIT, Master's Thesis 1999.

[96] Goh E.-J., Shacham H., Boneh D., "SiRiUs: securing remote untrusted storage," *Network and Distributed Systems Security (NDSS) Symposium*, pp. 131-145, 2003.

[97] Harrington A., Jensen C., "Cryptographic access control in a distributed file system," *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 158-165, 2003.

[98] Fu K. E., "Integrity and access control in untrusted content distribution," Massachusetts Institute of Technology (MIT), PhD Thesis 2005.

[99] Rajan R., "Efficient and privacy preserving multi user keyword search for cloud storage services," *In IJATER*, pp. 48-51, 2012.

[100] López-Alt A., Tromer E., Vaikuntanathan V., "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," *In STOC*, pp. 1219–1234, 2012.

[101] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," *Federal Information Processing Standards Publication 197*, 2001.

[102] Ateniese G., Benson K., Hohenberg S., "Key-Private Proxy Re-encryption," *In Proc. of RSA Conference on Topics in Cryptology: Cryptographers' Track*, pp. 279-294, 2009.

[103] Green M., Ateniese G., "Identity-Based Proxy Re-encryption," *In Proc. of the 5th International Conference on Applied Cryptography and Network Security (ACNS)*, pp. 288-306, 2007.

[104] Blaze M., Bleumer G., Strauss M., "Divertible Protocols and Atomic Proxy Cryptography," *In Proc. of Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques*, 1998.

[105] Hopcroft J. E., Motwani R., Ullman J. D., *Introduction to Automata Theory, Languages, and Computation (2 edition).*: Addison Wesley, 2001.

[106] Thompson K., "Programming Techniques: Regular Expression Search Algorithm," *Communications of the ACM 11*, pp. 419–422, 1968.

[107] Michael S., *Introduction to the Theory of Computation [Section 1.2, Theorem 1.19], page 55*.

[108] Goldreich O., *Foundations of Cryptography: Basic applications.*: Cambridge University, 2004.

[109] Stallings W., *Cryptography and Network Security: Principles and Practice.*: Pearson; 6 edition, 2013.

[110] Frakes, W. B., *Stemming algorithms, Information retrieval: data structures and algorithms.*: Upper Saddle River, NJ: Prentice-Hall, Inc, 1992.

APPENDIX A. GLOSSARY OF ACRONYMS

API – Application Programming Interface

ASP – Application Service Provider

CRA – Credit Report Agency

DFA – Deterministic Finite Automata

DaaS – Data as a Service

GUI – Graphical User Interface

NDFA – Nondeterministic Finite Automata

OOP – Object Oriented Programming

ORAM – Oblivious RAM (random-access memory)

OT – Oblivious Transfer

PPT – Probabilistic polynomial-time

PRF – Pseudo-Random Function

SaaS – Software as a Service

SE – Searchable Encryption

SFE – Secure Function Evaluation

SPKE – Searchable Public-Key Encryption

SPM – Secure Pattern Matching

SSE – Searchable Symmetric Encryption

TTP – Trusted Third Party

WB – White Box

WBC – White Box Cryptography