Stefan Bäuchl, BSc

A Formal and Experimental Evaluation of Searchable Symmetric Encryption Schemes

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Institute of Applied Information Processing and Communications

Dipl.-Ing. (FH) Dr.techn. Daniel Slamanig

Graz, March 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract (English)

With the rising popularity of cloud storage services, new challenges on securing and processing outsourced data arise. Due to security demands outsourced data needs to be encrypted. But all well known search algorithms are not suitable to operate on encrypted data. In recent years, different searchable encryption methods allowing the cloud provider to execute user generated search queries without learning critical information about the contents have been proposed. In this thesis, a comprehensive survey of Searchable Symmetric Encryption (SSE) constructions being the currently most efficient searching method of the ones providing reasonable security guarantees is provided. For this task, several state-of-the-art SSE schemes selected in a way to achieve best possible diversity are analysed with focus on security and practical usability. To evaluate their practical performance, all constructions have been implemented and extensively tested using realistic settings. Based on all findings, usage recommendations that can be seen as a guideline for typical scenarios are derived. The results reveal the different characteristics of schemes and show their suitability for practical usage, especially if the usage scenario can be well defined.

Key words: keyword search on encrypted data, searchable encryption, searchable symmetric encryption

Abstract (German)

Mit der stetig steigenden Popularität von Cloud Datenspeicherdiensten entstehen neue Herausforderungen hinsichtlich der Verarbeitung von sicher ausgelagerten Daten. Aufgrund der Tatsache, dass ausgelagerte Daten verschlüsselt werden müssen, können alle etablierten Suchalgorithmen für Klartextsuche nicht angewendet werden. In den letzten Jahren sind verschiedene Verfahren zum Durchsuchen von verschlüsselten Inhalten entwickelt worden, welche es dem Cloudanbieter ermöglichen, benutzergenerierte Suchanfragen durchzuführen ohne jegliche vertrauliche Informationen über den Inhalt zu erhalten. Diese Arbeit umfasst eine umfangreiche Studie der sogenannten Searchable Symmetric Encryption (SSE) Konstruktionen, die unter den jenen Verfahren welche effizient genug für praktische Anwendungen sind als sicherste Suchmethode gelten. Für diese Aufgabenstellung wurden verschiedene moderne, möglichst unterschiedliche SSE Verfahren ausgewählt und anhand der Schwerpunkte Sicherheit und Praxistauglichkeit analysiert. Um deren praktisches Leistungsvermögen zu evaluieren, wurden alle Konstruktionen implementiert und unter der Berücksichtigung von realistischen Anwendungsfällen ausgiebig getestet. Basierend auf den theoretischen und experimentellen Erkenntnissen sind für typische Anwendungsfälle Empfehlungen für die Verwendung von SSE Konstruktionen erstellt worden. Die Resultate unterstreichen die typischen Merkmale der Konstruktionen und zeigen deren Praxistauglichkeit auf, besonders unter genauer Definition des Einsatzgebietes.

Schlüsselwörter: Suchen auf verschlüsselten Daten, searchable encryption, searchable symmetric encryption

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Motivation

Over the last decades, information technologies have established their place in daily routines of people. While typical average consumer devices have been unconnected a while ago, nowadays a continuously rising capability of network bandwidth and computational power lead to highly connected devices and the possibility to share resources with minimal efforts.

Emerging from improved possibilities of using information technology environments, cloud computing and especially its usage to outsource data gained pervasive practice. As a consequence of the outsourced data, the cloud provider gains full access to all data leading to potential information leakage. To protect the privacy of the outsourced data, it has to be encrypted. However, this prevents the use of all well known search algorithms for unencrypted data.

In search for a solution, searchable encryption techniques that allow searching on encrypted data by using additional information provided by the user have been proposed.

As motivation for the practical relevance of searchable encryption, CryptDB [Pop+11] and other similar approaches such as Cipherbase [Ara+13] and Google's Encrypted BigQuery[1] have to be mentioned. The idea behind CryptDB is to search on encrypted databases using standard Structured Query Language (SQL) queries allowing the usage of existing applications without the need of adaptations. For this task, CryptDB acts as a proxy between application and database management system securing the queries. Depending on the SQL operations, a lot of different cryptographic approaches mostly based on Property-Preserving Encryption (PPE) are used, e.g. deterministic encryption, Order-Preserving Encryption (OPE), homomorphic encryption. To support searches on encrypted text such as the "like" operator, the searchable encryption approach from [SWP00] has been adapted.

Coming back to the general ideas of searchable encryption, different methods varying in features and security level have been proposed over the past few years. First of all, Oblivious Random Access Machines (ORAMs) [GO96; Nav15] offer a secure and theoretical efficient solution, but their practical performance can not compete with special purpose algorithms. Secondly, Functional Encryption (FE) defining a generalisation of asymmetric encryption can be used to build searchable encryption schemes [BSW11],

[1] Google Encrypted BigQuery: An experimental version of the BQ client which adds client-side encryption. URL: `https://github.com/google/encrypted-bigquery-client` (Accessed 10. December 2016)

but is not as efficient as constructions solely designed for searchable encryption. Fully Homomorphic Encryption (FHE) [Gen09] supporting arbitrary computations on ciphertexts using encrypted inputs is another theoretically applicable approach, but the efficiency is not suitable for practical usage. Asymmetric Searchable Encryption offers a specific searchable encryption solution, where everyone in possession of the public key can add documents but only the owner of the private key can generate search queries. Several popular Asymmetric Searchable Encryption schemes do exist [Bon+04; BSS08], but they are not as efficient as symmetric schemes. In modern Searchable Symmetric Encryption (SSE) schemes pictured in Figure 1.1, typically the user calculates a metadata file called index that is uploaded to the cloud storage together with the encrypted document collection. Later on, the cloud provider is able to execute user generated search queries without learning anything critical about its content. These constructions offer a practical tradeoff between performance and security. So far, no successful attacks exploiting the introduced information leakage without using additional knowledge are publicly known, as the later presented attacks on SSE schemes will indicate.

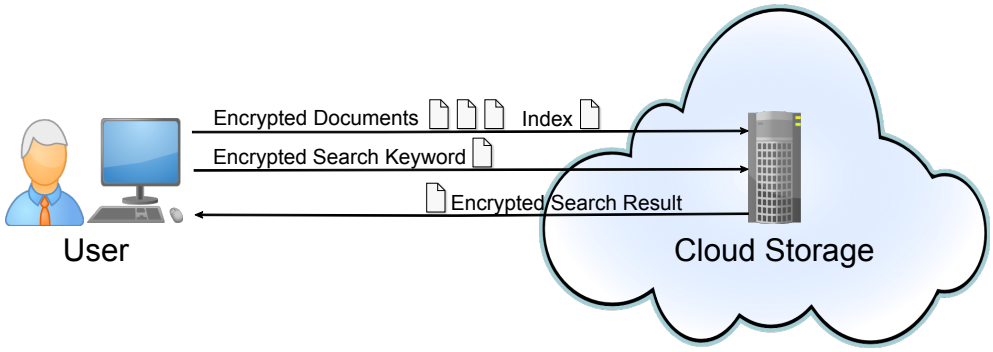


Figure 1.1.: Searchable Symmetric Encryption Setting.

The aim of this thesis is to provide a comprehensive overview of state-of-the-art SSE constructions with focus on security and practical usability. Therefore, security models for modeling the security guarantees of SSE schemes are discussed and a fundamental theoretical analysis and practical performance evaluation of selected SSE schemes is provided. The included SSE schemes have been selected by their usability in practice and their diversity to underline the characteristics of different construction methods.

For measuring their performance in practice, all selected schemes have been implement. To the best of our knowledge, no other implementations of SSE schemes apart from two fulltext searches[2] were publicly available at the time the implementation was done. At the time of writing, another implementation of partly different SSE schemes was published[3].

1.2. Related Work

Apart from the SSE schemes covered in this thesis [Goh03; CM05; Cur+06; KP13; Cas+13], other mentionable constructions exist [SWP00; KPR11; KPR12; Oga+13; SPS14; Cas+14; NPG14]. The full-text scheme from Song et al. [SWP00] was the first published searchable encryption scheme. The main idea is to construct a ciphertext by encrypting each word and XORing the encrypted word with

[2]Symmetric Searchable Encryption Library in Pure Java. URL: `https://github.com/sashank/jsse` (Accessed 10. December 2016)
[3]The Clusion Library. URL: `https://github.com/orochi89/Clusion` (Accessed 10. December 2016)

a pseudorandom bit stream relying on hash values. To search for a word, each document has to be sequentially scanned resulting in a search time linear in the total number of words. This construction does not achieve an appropriate SSE security notion as considered by state-of-the-art schemes.

Following the idea of using inverted indices in form of obfuscated linked lists stored in scrambled order proposed in [Cur+06], a lot of similar constructions based on analogous methods were suggested. In [Oga+13], Ogata et al. presented a space improved version of the non-adaptive scheme from [Cur+06] by introducing the additional leakage of the total number of keywords in the document collection.

The work of Kamara et al. in [KPR12] can be seen as a precursor of their subsequently suggested scheme in [KP13] following the same idea of providing a dynamic version of the scheme in [Cur+06]. In [KPR12], the authors tried to use same data structure as in [Cur+06] to construct a dynamic solution resulting in an extraordinary complex scheme. The same fundamental ideas have been applied in [KP13] presenting a much simpler but similar efficient construction which will be covered in this thesis.

The scheme proposed in [Cas+14], which can be seen as the successor of the scheme from [Cas+13] analysed in this thesis, uses a dictionary based data structure to store entries for each encrypted keyword-document pair identified by a pseudorandom label. To optimise the search efficiency for dictionaries stored on disk, the construction can be improved by partitioning the document identifiers associated to keywords into a fixed number of blocks and encrypting the blocks instead of encrypting each identifier individually. Additionally, for highly frequent keywords, the encrypted blocks are stored into an array and corresponding pointers are inserted into the dictionary. In comparison to [Cas+13], boolean queries are not supported straight away, though the scheme could be extended with the ideas from [Cas+13] to provide the same functionality.

In [NPG14], Naveed et al. proposed a scheme based on an update-supporting construction called Blind Storage, where each document is split into blocks stored at pseudorandom locations. In the actual scheme, instead of inserting the documents, for each keyword in the document collection a plaintext forward index is generated which is inserted into the Blind Storage. This scheme can be compared to the one in [KPR12] both proven to be secure in an adaptive setting, but in [NPG14] a better single keyword search performance is achieved.

Another different approach was presented in [SPS14] using an update-supporting data structure organised in levels of hash tables. Each level increases in size storing a maximum amount of entries, where each entry consists of an obfuscated keyword-document pair and additional information to support updates and efficient searching. Additionally, all levels are sorted by keywords using oblivious sorting and conditions on which level entries are inserted exist. Consequently, this scheme provides fast single keyword searching, but does not support multiple keyword queries as the scheme in [Cas+13].

As extension of the classical SSE requirements covered in later parts of this thesis, Bost et al. [BFP16] proposed a verifiable SSE scheme allowing the user to verify that the search result is correct, i.e., no wrong files are included and no files are missing. The construction follows the ideas from [SPS14], but instead of using conventional hash tables, a modified construction called verifiable hash table storing additional information for each entry is applied. Additionally, authenticated encryption is used to encode all entries. Compared to [SPS14], the overall construction achieves similar asymptotic performance, but no experimental results are provided to verify that the additional operations do not affect the practical performance too much.

As one of the representatives of multi-user searchable encryption schemes, the scheme from [RMÖ15] has to be mentioned. In a multi-user setting, the data owner can grant search permission for specific documents to other users. In the construction from [RMÖ15] a proxy, which does not have to be trustworthy, is used to transform a user generated search query into Private Information Retrieval (PIR) [Cho+95] queries using bilinear pairings. While the authors do not provide experimental implementation results,

the requirement for pairing computations is expected to result in a significant slower practical performance compared to usual SSE schemes.

In [Ish+16], Ishai et al. proposed a construction in a distributed model where two servers are used. In their construction a B-tree is used as data structure which is stored at one of the servers. To search for a keyword, the client interacts with both servers to iterate trough the tree. By applying secret sharing techniques, the data is randomly permuted and none of the participants learns the path that has been taken. After recovering the tree, PIR queries are used to retrieve the data. Even though the authors included experimental evaluation results, no practical performance comparison to conventional SSE constructions is provided leading to the assumption that SSE schemes could be superior in practice.

It needs to be mentioned that searchable encryption is a very active research field so that we do not claim to provide an exhaustive overview over all existing approaches. However, we have tried to provide a comprehensive overview.

1.3. Outline of this Thesis

The remainder of this thesis is structured as follows: Chapter 2 introduces basic mathematical foundations, a brief introduction to cryptography, symmetric and asymmetric encryption. Moreover, it includes a survey of popular security models used to define the security guarantees of modern encryption schemes. Finally, a description of hash functions and the probabilistic data structure called Bloom filter closes the overview of basic cryptographic concepts.

Chapter 3 motivates the usability of searchable encryption, in particular of SSE schemes, and provides a categorisation of these schemes. After providing a mathematical notation and defining an abstract, unified index-based scheme, state-of-the-art security models for SSE schemes are discussed.

Chapter 4 provides the theoretical part of this thesis in form of an analysis of selected SSE schemes. For all schemes, a detailed description is presented, followed by an extensive analysis focused on information leakage and asymptotic performance.

Chapter 5 describes the practical part of the work providing noteworthy details about the implementation of all selected SSE schemes. Subsequently, Chapter 6 provides a comprehensive evaluation of the schemes implementation performance underlining the findings of the theoretical analysis.

Chapter 7 includes a guideline for using SSE schemes based on the theoretical and practical findings. For certain typical situations, recommendations to select the suitable SSE scheme are provided.

Finally, Chapter 8 presents a conclusion and directions for further work, which closes this thesis.

Chapter 2

Preliminaries

This chapter provides an overview of some basic cryptographic primitives and concepts needed to understand the searchable encryption algorithms presented in later parts of this thesis. Readers that are already familiar with basic cryptographic concepts including mathematical foundations and security definitions might skip this chapter.

2.1. Mathematical Foundations

In this thesis, sampling an element x from a set $\mathcal{X}$ uniformly is denoted by $x \xleftarrow{\$} \mathcal{X}$. Given an algorithm $\mathtt{A}$, its output a is defined as $a \leftarrow \mathtt{A}$. The concatenation of two values x, y is given by $x||y$, a bitwise OR is defined as $x \mid y$ and a bitwise XOR is denoted by $x \oplus y$. Given a set $\mathcal{S}$, the number of elements in $\mathcal{S}$ is defined as $|\mathcal{S}|$. Given a matrix M, the element in the i^{th} row and j^{th} column is addressed as $M[i, j]$. A security parameter defining the input size of an algorithm in unary representation is denoted as k, a key is denoted as K. In mathematical definitions and security proofs, one often uses unspecified polynomial functions. In this thesis, complete unspecified polynomial functions are indicated by $\mathsf{poly}(\cdot)$, where $\mathsf{poly}(x)$ indicates an arbitrary polynomial function in one variable x.

Definition 2.1 (Group [Mao03])**.** *A group $(\mathcal{G}, \circ)$ is a set $\mathcal{G}$ and a binary operation $\circ$ satisfying the following conditions:*

- *Closure: For all $x, y \in \mathcal{G} : x \circ y \in \mathcal{G}$.*
- *Associativity: For all $x, y, z \in \mathcal{G} : x \circ (y \circ z) = (x \circ y) \circ z$.*
- *Identity: There exists a unique identity $e \in \mathcal{G}$ such that for all $x \in \mathcal{G} : e \circ x = x \circ e = x$.*
- *Inverse: For all $x \in \mathcal{G}$ there exits an inverse $x^{-1} \in \mathcal{G} : x \circ x^{-1} = x^{-1} \circ x = e$.*

For better readability, a group $(\mathcal{G}, \circ)$ is often denoted by $\mathcal{G}$. A group $\mathcal{G}$ is called finite if it contains a finite number of elements, where the number of elements is called the order of the group denoted by $|\mathcal{G}|$. As widely used example, the set $\mathbb{Z}_n = \{0, \ldots, n-1\}$ with integer $n > 1$ is a finite group under addition modulo n.

Definition 2.2 (Abelian Group [Mao03]). *A group $(\mathcal{G}, \circ)$ is called abelian if the commutativity axiom holds:*

- *Commutativity: For all $x, y \in \mathcal{G} : x \circ y = y \circ x$.*

Definition 2.3 (Cyclic Group [Mao03]). *A group $(\mathcal{G}, \circ)$ is called cyclic if there exists an element $g \in \mathcal{G}$ denoted as generator $\langle g \rangle$ such that for every element $x \in \mathcal{G}$, there exists an $i \in \mathbb{Z}$ such that $x = g^i$.*

As important example, $\mathbb{Z}_p^*$ being a multiplicative, cyclic, abelian group of integers modulo prime p is of major interest in cryptography.

Definition 2.4 (Field [HMV04]). *A field $(\mathcal{S}, +, \cdot)$ is a set $\mathcal{S}$ with two binary operations "$+$" (addition) and "$\cdot$" (multiplication) satisfying the following properties:*

- *$(\mathcal{S}, +)$ is an additive abelian group with identity 0.*
- *$(\mathcal{S}, \cdot)$ is a multiplicative abelian group with identity 1.*
- *Distributivity: For all $x, y, z \in \mathcal{S} : x \cdot (y + z) = x \cdot y + x \cdot z$.*

It is a convenience to write $\mathbb{F}$ to denote a field. A field is called finite field if it contains a finite number of elements and the order of a finite field is always a power of a prime. Consequently, a finite field $\mathbb{F}_q$ of order q exists if $q = p^r$ with prime p known as characteristic of $\mathbb{F}_q$. Additionally, if $r = 1$, $\mathbb{F}_q$ is called prime field, otherwise for $r \geq 2$ $\mathbb{F}_q$ is known as extension field.

In cryptography, prime fields $\mathbb{F}_p$ and binary extensions fields $\mathbb{F}_{2^m}$, which can be constructed using a polynomial basis, are widely used.

Definition 2.5 (General Elliptic Curve [HMV04]). *An general elliptic curve E over an arbitrary field $\mathbb{F}$ is a smooth curve defined by the affine Weierstrass equation*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.1}$$

with coefficients $a_1, a_2, a_3, a_4, a_5, a_6 \in \mathbb{F}$.

In literature, the notation $E/\mathbb{F}$ is prevailing to emphasise that E is defined over $\mathbb{F}$. The notation $E(\mathbb{F})$ defines the set of all points $(x, y) \in \mathbb{F}^2$ that satisfy the Weierstrass equation together with the additional point ∞, which is the point at infinity serving as identity of the abelian group $E(\mathbb{F})$. Depending on the type of curve, there exist a lot of different approaches and ongoing researches on computing the basic curve operations (point addition, point doubling) and their usage in point addition and scalar multiplication efficiently.

In later parts of this thesis, elliptic curves over prime fields will be used. Consequently, a simplified form of the Weierstrass equation can be used to describe such special forms of elliptic curves for prime fields[1] $\mathbb{F}_p$ with $p > 3$.

Definition 2.6 (Elliptic Curve Over Prime Fields [HMV04]). *An elliptic curve E over a field $\mathbb{F}$ with characteristic $\neq 2$ or 3 is defined by the equation*

$$E : y^2 = x^3 + ax + b, \tag{2.2}$$

where $a, b \in \mathbb{F}$ are constants with $4a^3 + 27b^2 \neq 0$, ensuring that the cubic on the right sight of Equation 2.2 has no repeating roots.

[1] Reader interested in simplified definitions for binary extension fields and more in-depth information on the simplified Weierstrass equation are referred to [Mil85; Kob87; HMV04].

Definition 2.7 (Elliptic Curve Discrete Logarithm Problem [HMV04])**.** *Let E be an elliptic curve over a finite field $\mathbb{F}_q$. Given a point $P \in E(\mathbb{F}_q)$ and a point $Q \in \langle P \rangle$, the Elliptic Curve Discrete Logarithm Problem (ECDLP) is defined as finding the smallest, positive integer x such that:*

$$Q = x \cdot P. \tag{2.3}$$

Since elliptic curves with points in $\mathbb{F}_p$ are finite groups [Kob87] where the ECDLP is hard, they can be used to implement asymmetric cryptography. For practical implementations, there exist several standards providing curves that aim to be secure for usage in elliptic curve cryptography (ECC) [LM10; Nat13].

Definition 2.8 (Probabilistic Polynomial Time [KL14])**.** *An algorithm $\mathtt{A}$ is said to be probabilistic polynomial-time (PPT) if $\mathtt{A}$ is allowed to make random choices and if the runtime is polynomial restricted in the length of the input $x \in \{0,1\}^*$, i.e., the runtime is bounded by $\mathsf{poly}(|x|)$.*

Due to the randomness involved in a PPT algorithm, it's output represents a random variable. In security proofs, the internal coin toss of a PPT algorithm is often not explicitly stated. On the other hand, if the randomness is defined by a random variable the algorithm receives as additional input, this random variable has to be covered.

In practice, the PPT property is often used to define real world adversaries because it is reasonable to assume that in real world settings, the resources are polynomial bounded and a certain probability for a successful attack exists. In the security proofs later presented in this thesis, the notation $\mathtt{A}^{\mathsf{func}(\cdot)}(x)$ defines a PPT adversary $\mathtt{A}$ receiving parameter x as input and having oracle access to function $\mathsf{func}(\cdot)$.

Definition 2.9 (Negligible Function [KL14])**.** *A function $\mathsf{f}(x)\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ with $\mathbb{R}_{\geq 0} = \{x \in \mathbb{R}\colon x \geq 0\}$ is called negligible if for every positive polynomial $\mathsf{poly}(\cdot)$, there exists a number $N \in \mathbb{N}$ such that for all $n > N$ with $n \in \mathbb{N}$:*

$$\mathsf{f}(x) < \frac{1}{\mathsf{poly}(n)}. \tag{2.4}$$

Intuitively, this functions can be used to describe security of cryptographic primitives. Considering a practical environment, it is very reasonable to assume that an adversary is polynomial bounded in all resources. Consequently, the probability of a successful attack has to be smaller than the inverse of an arbitrary polynomial. Obviously, this property corresponds to the definition of negligible functions predestining them for security notations.

Definition 2.10 (Keyed Function [KL14])**.** *A keyed function $\mathsf{F}\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$ is a function with two inputs, where the first input is called key.*

The function F is called efficient if given a key $K \in \{0,1\}^k$ and an input message $M \in \{0,1\}^*$, $\mathsf{F}(K,M) = \mathsf{F}_K(M)$ can be computed in polynomial time.

Definition 2.11 (Pseudorandom Function [KL14])**.** *Let $\mathsf{F}_K : \{0,1\}^m \to \{0,1\}^c$ be an efficient keyed function, U_f be the set of all functions mapping $\{0,1\}^m$ to $\{0,1\}^c$ and $\Pr$ be the probability function. The function F_K is pseudorandom if for any PPT distinguisher $\mathtt{D}$, there exists a negligible function $\mathsf{negl}(k)$ such that*

$$\left| \Pr_{K \xleftarrow{\$} \{0,1\}^k} \left[\mathtt{D}^{\mathsf{F}_K(\cdot)}(1^k) = 1 \right] - \Pr_{f \xleftarrow{\$} U_f} \left[\mathtt{D}^{f(\cdot)}(1^k) = 1 \right] \right| \leq \mathsf{negl}(k). \tag{2.5}$$

Informally, an adversary has to distinguish between a pseudorandom function (PRF) and a function selected at random from the set of all functions having the same domain denoted as U_f. As noted by [KL14], it is important to see that the PPT distinguisher, i.e., the adversary, is not given the secret key K but only the output of the respective functions. By definition, the probability that the adversary is able to guess correctly has to be negligible. Consequently, PRFs are functions that are indistinguishable from truly random functions given a polynomial time bounded adversary. In practice, PRFs can be constructed utilising block ciphers or hash functions [Hal+98].

Definition 2.12 (Pseudorandom Permutation [Hal+98])**.** *A pseudorandom permutation (PRP) is a special form of a PRF where the mapping is a bijection with identical domain and co-domain.*

Following the same reasoning as in the PRF definition, PRPs are permutations that are indistinguishable from truly random permutations given a polynomial time bounded adversary [KL14]. To provide some practical realisations, choosing a block cipher in suitable mode or utilising a PRF in multiple rounds are some possible construction methods [Hal+98; KL14].

Since a basic mathematical foundation has been defined, some fundamentals of modern cryptographic can be discussed.

2.2. Introduction to Cryptography

Going way back to the ancient world, cryptography is known as the science of protecting information. The cryptography used a few thousand years ago was based on simple methods to transport information, for example about troop movement and war plans. Over the last decades, information technologies have established their place in daily routines of people. Due to the continuously rising capability of the pervasive computational power, more complex methods are needed to ensure the confidentiality of data. Before discussing some of these modern cryptographic approaches, it is necessary to define a general cryptographic terminology based on the works in [MOV96; Mao03; KL14].

To start with, encryption is the procedure of transforming data in an obfuscated unreadable form which should be indistinguishable from random data. The input data which shall be encrypted is called plaintext, while the resulting output data of the encryption process is called ciphertext. As a counterpart, decryption uses ciphertext as input and transforms it back to the original form, i.e., the plaintext. The set of all encryption and corresponding decryption functions is called cipher and for both encrypting and decrypting, an additional secret information called key is needed. Furthermore, Kerckhoffs' principle, stating that the security relies only on the secrecy of the key and not the algorithm itself, has to be applied.

All these components are needed to provide an expedient cryptosystem. The aim of such a system is to provide data confidentiality by encrypting the data in order to make it incomprehensible for people which are not in possession of the key needed to decrypt it.

In modern cryptography, a distinction between symmetric and asymmetric encryption can be made. While asymmetric encryption schemes need different keys for encrypting and decrypting data, in symmetric cryptography algorithms the decryption key is either efficiently computable given the encryption key, or equal to the encryption key [MOV96].

2.3. Symmetric Encryption

Definition 2.13 (Symmetric Encryption Scheme [KL14])**.** *A symmetric or private key encryption scheme is a tuple of polynomial-time algorithm* $(\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ *such that:*

$\texttt{Gen}(1^k)$*: The probabilistic key-generation algorithm* Gen *takes a security parameter k in unary representation as input and computes a random secret key K.*

$\texttt{Enc}_K(M)$*: The possibly probabilistic encryption algorithm* Enc *receives a secret key K and a plaintext message M and outputs a ciphertext C.*

$\texttt{Dec}_K(C)$*: The deterministic decryption algorithm* Dec *takes a secret key K and a ciphertext C and outputs a plaintext message. In case of a decryption error, the algorithm outputs a generic error symbol* $\perp$*.*

Additionally, for all secret parameters k, for all keys $K \leftarrow \texttt{Gen}(1^k)$ *and for all plaintext messages M, it is required that* $\texttt{Dec}_K(\texttt{Enc}_K(M)) = M$*.*

One of the main advantages of symmetric encryption schemes is the high performance of these types of ciphers. Compared to the asymmetric ones, symmetric ciphers are several orders of magnitude faster making them suitable for operating on large amounts of data [MOV96]. Since many secret key encryption schemes are built upon mathematical functions that can be implemented efficiently in software, these schemes are very versatile usable, for example in searchable encryption schemes.

Another benefit of the symmetric approach are the significant smaller key sizes compared to asymmetric ciphers without loss of security. According to several guidelines [Bab+12; Nat16], for modern symmetric block ciphers a key-length of 128 bits or more is recommended, while the key-length of modern asymmetric ciphers should be at least 2048 bits, depending on the concrete setting. For elliptic curves, a key-length of 224 bits ore more is recommended [Nat16].

In symmetric cryptography, for each pair of participants an unique key is needed. As a result, the number of keys required grows quadratic in the number of participants which makes the usage of symmetric cryptography for a group of users impractical very soon. Undoubtedly, distributing and managing the keys are the main challenges of symmetric cryptography.

Examining the construction methods of symmetric ciphers, stream and block ciphers are established building blocks.

2.3.1. Stream Cipher

In a symmetric stream cipher, the plaintext is encrypted bit by bit. The basic idea is to calculate the XOR of a pseudo-random key stream and the plaintext. To retrieve the plaintext, the ciphertext has to be XORed with the key stream. The generation of the pseudo-random key stream depends on the type of stream cipher. In a synchronous stream cipher, the key stream is only based on the secret key, while in the asynchronous case the last few ciphertext bits are also utilised. Since synchronous stream ciphers will always produce identical key streams for the same key, the key has to be changed as often as possible to avoid correlation attacks.

Regarding the usage of stream ciphers in practice, it has to be mentioned that it is possible to build stream ciphers based on block ciphers. According to [KL14] and as an outlook to block ciphers, the confidence in security of dedicated stream ciphers appears to be lower in comparison to block ciphers in practice. Consequently, Katz et al. recommend the usage of block ciphers in a suitable mode of operation if possible.

2.3.2. Block Cipher

While stream ciphers apply bit by bit encryption, block ciphers are defined upon plaintext blocks of fixed size and encrypt each block at once. Therefore, the plaintext has to be split into blocks before applying the cipher. If the length of the plaintext is not a multiple of the block size, a padding scheme can be applied to extend the input to a multiple of the block size.

Given a plaintext, there are different possibilities how a block cipher can be used to compute the ciphertext. The possible methods of processing single blocks to produce the ciphertext are called modes of operation. A mode of operation defines how a block cipher is used to encrypt or decrypt an arbitrary number of blocks.

In the following and based upon a recommendation report provided in [Nat01b], a short overview of the most common modes for modern block ciphers is provided.

Electronic Codebook Mode: The Electronic Codebook (ECB) mode is the most intuitive and simplest possible mode. The plaintext is split into blocks and each block is encrypted independently using the same key. Since this mode maps identical plaintext blocks to identical ciphertext blocks, it is highly recommended to never use this mode.

Cipher Block Chaining Mode: In the Cipher Block Chaining (CBC) mode, each plaintext block is XORed with the previous ciphertext block before the encryption. For the first ciphertext block, an initialisation vector (IV) is used to compensate the non-existing previous ciphertext block.

Cipher Feedback Mode: In the Cipher Feedback (CFB) mode, each plaintext block is XORed with the encryption of the previous ciphertext block using an IV for the first encryption block. Therefore, this mode generates a key stream using the last few ciphertext bits that is XORed with the plaintext. Consequently, the CFB allows operating the block cipher as asynchronous stream cipher.

Output Feedback Mode: In the Output Feedback (OFB) mode, each plaintext block is XORed with the encryption of the previous encryption output using an IV to compensate the non-existing first encryption result. Therefore, this mode generates a key stream based solely on the IV and the key. Consequently, the OFB mode allows operating the block cipher as synchronous stream cipher.

Counter Mode: In the Counter (CTR) mode, an initial counter value is encrypted and XORed with the first plaintext block to receive the first ciphertext block. For each upcoming plaintext block, the counter value is increased by one and the same steps are applied. Similar to the OFB mode, a key stream is generated without using previous ciphertext bits. Therefore, the CTR mode acts as synchronous stream cipher.

The following review of the currently most popular symmetric cipher completes the survey about symmetric encryption.

2.3.3. Advanced Encryption Standard

In 1997, the National Institute of Standards and Technology (NIST) announced the Advanced Encryption Standard (AES) competition to find a successor of the outdated Data Encryption Standard (DES) [DR02]. In the following, the Rijndael cipher, which was declared as AES after several selection rounds, is briefly introduced based on the developer's Rijndael design book [DR02] and the standardisation publication [Nat01a].

While the original Rijndael cipher was specified upon block sizes being a multiple of 32 bits at a minimum of 128 bits, the AES is standardised for a fixed block size of 128 bits. The key size of the original Rijndael

cipher could be specified independently of the block size, in the AES specification key lengths of 128, 192 and 256 bits are defined.

Considering the construction method of the AES, it is an iterative block cipher. These types of ciphers apply simple operations multiple times on the plaintext in order to compute the ciphertext, where the simple operations are known as round functions or round transformations. Consequently, a single execution of a round function is called round or cycle, where for each one a round key is needed. The round keys are generated by a key scheduling algorithm, which is part of the cryptosystem.

In case of the AES, the key size determines the number of rounds that have to be processed in order to encrypt or decrypt a given input. Accordingly, Table 2.1 defines all possible combinations of key size and number of rounds.

	Key Size (Bits)	Block Size (Bits)	Number of Rounds
AES-128	128	128	10
AES-192	192	128	12
AES-256	256	128	14

Table 2.1.: AES: Key Size and Number of Rounds Combinations.

Without going into details, each round - except the last one - consists of four transformations called steps:

SubBytes: Each input byte is substituted by another byte obtained from a lookup table called S-box.

ShiftRows: Some bytes of the intermediate value are shifted cyclically to the left.

MixColumns: The input bytes are grouped to blocks of four bytes which are multiplied by a fixed value.

AddRoundKey: The round key is XORed to the intermediate result.

In the last round, the MixColumns step is skipped and before the first round, an additional AddRoundKey step is performed. Obviously, the round keys have to be computed by the key scheduling algorithm before the initial round. All steps are invertible, hence, in the decryption process the inverted functions are applied. To finalise the high-level description of the AES, a summary of the whole r round enryption process is given:

1. Key Schedule: Generate Round Keys
2. Initial Round: AddRoundKey
3. Rounds $1, \ldots, r-1$: SubBytes, ShiftRows, MixColumns, AddRoundKey
4. Final Round r: SubBytes, ShiftRows, AddRoundKey

To close the survey of the AES, it is fair to say that the AES provides very strong security guarantees. While there exist attacks given certain assumptions or attacks being slightly better than exhaustive key searches [BKR11], there are no known attacks that affect the practical security of the AES.

All in all the standardised AES algorithm provides strong security and performance [Sch+99; DPR00] if applied correctly, making it the currently most popular block cipher.

2.4. Asymmetric Encryption

In asymmetric encryption schemes, two different keys for encrypting and decrypting data are needed. The encryption key is called public key K_{pub}, the decryption key is called private key K_{priv} and the combination

of corresponding encryption key and decryption key is known as key pair (K_{pub}, K_{priv}) [MOV96; KL14]. The private and public key are related mathematically, but it should be infeasible in practice to calculate the corresponding private key K_{priv} of a given public key K_{pub}.

Definition 2.14 (Asymmetric Encryption Scheme [KL14])**.** *An asymmetric or public key encryption scheme is a tuple of polynomial-time algorithm* $(\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ *such that:*

$\texttt{Gen}(1^k)$*: The probabilistic key-generation algorithm* Gen *takes a security parameter k as input and computes a key pair* (K_{pub}, K_{priv}).

$\texttt{Enc}_{K_{pub}}(M)$*: The probabilistic encryption algorithm* Enc *receives a public key* K_{pub} *and a plaintext message M and outputs a ciphertext C.*

$\texttt{Dec}_{K_{priv}}(C)$*: The deterministic decryption algorithm* Dec *takes a private key* K_{priv} *and a ciphertext C and outputs a plaintext message. In case of a decryption error, the algorithm outputs a generic error symbol* $\perp$.

Additionally, it is required that $\texttt{Dec}_{K_{priv}}(\texttt{Enc}_{K_{pub}}(M)) = M$ *for all valid messages M and key pairs* (K_{pub}, K_{priv}) *generated by* $\texttt{Gen}(1^k)$.

Since the public key is only used for the encryption but not the decryption process, knowledge of the public key does not affect the confidentiality of the message in any way. If the data shall be decrypted, the corresponding private key has to be used. Therefore, only the confidentiality of the private key has to be protected.

One of the main advantages of asymmetric cryptography is that one key pair per participant is sufficient, while $\binom{n}{2}$ are needed for symmetric cryptography. Another advantage of asymmetric cryptography is that the key pair can be used for a long period without security concerns. While symmetric keys need to be changed frequently, for example for each file or session, asymmetric keys can often be used for years [Nat16].

The main disadvantage of asymmetric cryptography is the much worse performance compared to the symmetric counterpart. Therefore, in practice symmetric encryption is used for larger amounts of data and asymmetric encryption is typically used to distribute a symmetric key.

For constructing asymmetric cryptosystems, special forms of one-way functions called trapdoor one-way functions are used. A trapdoor one-way function is, just like a general one-way function, easy to compute and computationally hard to invert, but given an extra secret information called trapdoor, the inversion can be computed efficiently.

In practice, a number of popular functions and resulting number theoretical problems are utilised for building asymmetric cryptosystems. To provide some examples, the integer factorisation problem is defined as finding the factors of a given integer. If only prime numbers are considered as factors, the problem is known as prime factorisation. Another famous example is the discrete logarithm problem already introduced in Section 2.1 for elliptic curve groups.

2.5. Security of Encryption Schemes

Instead of relying on security through obscurity, cryptographic systems need to be designed accordingly to Kerckhoffs' principle [Mao03; Nat08; KL14].

2.5.1. Types of Attacks

Regardless of the type of cryptosystem, attacks on cryptosystems can be classified into several categories. The following overview of the most common attack types – ordered by powerfulness starting from the weakest to the strongest one – is based on [KL14].

Ciphertext only: To begin with the simplest type of attack, in a ciphertext only attack the adversary can only observe one or more ciphertexts. If an attacker should be able to gain information about the decryption key or the plaintext, the underlying encryption scheme has to be classified as completely insecure [MOV96].

Known plaintext: In this type of attack, the adversary is in possession of one or more pairs of ciphertext and corresponding plaintext. Using this information, the attacker tries to gain information about the decryption key or some other plaintext which has been encrypted with the same key. Obviously, this attack is only meaningful in the symmetric setting.

Chosen plaintext: In the chosen-plaintext attack (CPA), the attacker has access to a cryptosystem and is able to chose arbitrary plaintexts and obtain the corresponding ciphertexts. As in the known plaintext attack, the adversary uses the available plaintext-ciphertext pairs to deduce information.

Chosen ciphertext: In this type of attack, the adversary has the same powers as in a CPA, but is additionally allowed to chose ciphertexts and retrieves the corresponding decrypted plaintexts by a decryption oracle. For security notations, a further distinction can be made whether the adversary is allowed to use the decryption oracle after the challenge ciphertext is created (adaptive chosen-chiphertext attack (CCA2)) or not (non-adaptive chosen-chiphertext attack (CCA1)). Again the attacker tries to gain information using the constructed plaintext-ciphertext pairs.

Regarding the security of a cipher, different notations and definitions exist. Historically, a cipher is called broken if an attack with better performance than an exhaustive key search exists. In an exhaustive key search, also known as brute force attack, the attacker tries to find the correct key by trying all possible ones.

2.5.2. Types of Security Definitions

Due to the lack of significance for modern encryption schemes, more meaningful security notations have been introducted. Since the majority of security definitions for modern cryptography are either simulation or game based, the general ideas behind this types of security definitions are summarised now.

Simulation Based Security Definitions: The general idea of this type of definition is to compare the information leakage in an ideal world with the real world. In the ideal world, the cryptographic primitive, for example the encryption scheme, is secure per definition. The common way for achieving this property is to avoid using the primitive in question at all, typically by providing random output instead of computed one. In the real world, the cryptographic primitive is used as intended. If the information learned in a real world setting is negligibly different to the ideal world, the cryptographic primitive is considered secure.

Game Based Security Definitions: These definitions do not compare ideal and real settings, but are rather based on interactions between two players known as adversary and challenger. As one would expect, the challenger creates a challenge involving the cryptographic primitive that shall be tested and the adversary wins the game if the correct output for the specific challenge can be provided. In this setting, the challenger has full access to all security parameters such as secret keys, while

the adversary can only access (public) functions and information provided by the challenger. If an adversary can not achieve a significant better probability of predicting or computing the correct output compared to a trivial adversary which is just guessing, the underlying cryptographic primitive is considered secure.

Based on these concepts, some of the currently most important security models and definitions for modern encryption schemes are discussed subsequently.

2.5.3. Perfect Secrecy

Definition 2.15 (Perfect Secrecy [KL14])**.** *Let RV_M be a random variable for plaintext messages and RV_C be a random variable for ciphertexts. An encryption scheme with messages space $\mathcal{M}$ is said to be perfectly secret or information-theoretically secure if for all probability distributions over $\mathcal{M}$, all messages $M \in \mathcal{M}$ and every possible ciphertext $C \in \mathcal{C}$, i.e., $Pr[RV_C = C] > 0$:*

$$\Pr[RV_M = M | RV_C = C] = \Pr[RV_M = M]. \tag{2.6}$$

From an information-theoretic viewpoint, perfect secrecy for an encryption scheme can be achieved if the random variables RV_M and RV_C are independent. The idea behind the given definition is that even with knowledge of the ciphertext, the adversary does not gain any information about the plaintext, i.e., the a priori knowledge is equal to the a posteriori knowledge. Intuitively, an encryption scheme achieves perfect secrecy if an attacker with unlimited computational power is not able to gain any useful information about the plaintext of a given ciphertext [MOV96; KL14]. According to Shannon's theorem, perfect secrecy can only be achieved if the size of key space $\mathcal{K}$ is at least as large as the message space $\mathcal{M}$ [KL14]. As a final note, perfect secrecy is equivalent to similar definition known as perfect indistinguishability [KL14].

In practice, a one-time pad could be used to achieve perfect secrecy. A one-time pad encryption is defined as the XOR of key and plaintext [KL14], where the key is a uniform string of the same length as the plaintext. Additionally, the key has to be used just once. Since this construction is usually impractical, security is relaxed to a computational one.

2.5.4. Semantic Security

An encryption scheme achieves semantic security, if any PPT attacker receiving a ciphertext is not able to learn any useful information about the plaintext[2] [Mao03]. Consequently, semantic security is the computational security equivalent to perfect secrecy since the attacker's resources are polynomially bounded instead of unlimited.

While semantic security can be defined using a simulation based approach, there exist (in dependence of the adversary's power, i.e., the attack type) equivalent game based definitions that are widely used and can be considered as the current standard definitions for describing the security of symmetric and asymmetric encryption schemes. To keep things simple and easier to understand, the following security models are presented for symmetric ciphers based on the works by [Mao03; KL14].

[2] As stated in [Mao03], the length of the corresponding plaintext is not considered an useful information.

2.5.5. IND-CPA

Definition 2.16 (IND-CPA)**.** *Let* $\Pi = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ *be an encryption scheme,* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *be a PPT adversary and* $k \in \mathbb{N}$ *be a security parameter. The game based experiment* $\mathsf{Ind\text{-}CPA}_{\Pi,\mathcal{A}}(k)$ *is defined as*

$\mathsf{Ind\text{-}CPA}_{\Pi,\mathcal{A}}(k)$
- $K \leftarrow \mathtt{Gen}(1^k)$
- $(st_{\mathcal{A}}, M_0, M_1) \leftarrow \mathcal{A}_1{}^{\mathtt{Enc}_K(\cdot)}$ *with* $|M_0| = |M_1|$
- $b \xleftarrow{\$} \{0,1\}$
- $C_b \leftarrow \mathtt{Enc}_K(M_b)$
- $b' \leftarrow \mathcal{A}_2{}^{\mathtt{Enc}_K(\cdot)}(st_{\mathcal{A}}, C_b)$
- *if* $b' = b$*: output* 1
- *else output* 0*.*

An encryption scheme $\Pi = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ *is said to achieve indistinguishability under chosen-plaintext attack (IND-CPA) if for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, there exists a negligible function such that*

$$\Pr\left[\mathsf{Ind\text{-}CPA}_{\Pi,\mathcal{A}}(k) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(k), \tag{2.7}$$

where the probabilities are taken over the randomness in $\mathcal{A}$*, the internal coin toss of the* `Gen` *and* `Enc` *algorithm and over the choice of b which is sampled uniformly from* $\{0,1\}$*.*

As formally defined in Equation 2.7, the encryption scheme used in the game based experiment achieves IND-CPA security if for all PPT adversaries, the probability that the adversary can guess correctly which of the adversarially chosen messages M_0 or M_1 has been encrypted is at most negligible better than random guessing. Informally, it defines that an adversary performing a CPA is not able to distinguish which of the chosen plaintexts has been encoded, i.e., the ciphertexts are indistinguishable. As stated in [Mao03], the provided definition of IND-CPA is equivalent to the definition of semantic security.

Taking a closer look at the Ind-CPA experiment, it can be seen that the adversary has oracle access to the encryption function. Hence, the adversary is allowed to make calls to the encryption oracle before the plaintext messages M_0 and M_1 have to be chosen. Therefore, the adversary could encrypt M_0, M_1 and test which of these ciphertexts matches C_b. As a consequence, the encryption scheme has to be randomised to achieve IND-CPA security. If the encryption algorithm `Enc` is randomised, the oracle returns different ciphertexts for identical plaintexts with overwhelming probability.

To provide a concrete example, the AES in CBC mode achieves IND-CPA security since a different IV is used for each encryption. On the other hand, the AES in ECB mode or the textbook Rivest Shamir Adleman (RSA) encryption scheme are not IND-CPA secure due to their deterministic nature. To achieve IND-CPA security for RSA, one needs to use a suitable probabilistic padding scheme.

For completeness and use in later parts of this thesis, it has to be mentioned that a very similar definition called pseudorandomness against chosen-plaintext attack (PCPA) exists. An encryption scheme achieves PCPA security, if it can not be distinguished whether a chosen plaintext or a random message has been encrypted. Consequently, PCPA is slightly stronger than IND-CPA. In practice, common IND-CPA secure encryption schemes like AES in CTR mode also achieve PCPA security [Cur+06].

2.5.6. IND-CCA

Definition 2.17 (IND-CCA2)**.** *Let* $\Pi = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ *be an encryption scheme,* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *be a PPT adversary and* $k \in \mathbb{N}$ *be a security parameter. The game based experiment* $\mathsf{Ind\text{-}CCA2}_{\Pi,\mathcal{A}}(k)$ *is defined as*

$\mathsf{Ind\text{-}CCA2}_{\Pi,\mathcal{A}}(k)$
$K \leftarrow \mathtt{Gen}(1^k)$
$(st_{\mathcal{A}}, M_0, M_1) \leftarrow \mathcal{A}_1^{\mathtt{Enc}_K(\cdot),\mathtt{Dec}_K(\cdot)}$ *with* $|M_0| = |M_1|$
$b \xleftarrow{\$} \{0,1\}$
$C_b \leftarrow \mathtt{Enc}_K(M_b)$
$b' \leftarrow \mathcal{A}_2^{\mathtt{Enc}_K(\cdot),\mathtt{Dec}_K(\cdot)}(st_{\mathcal{A}}, C_b)$
if $b' = b$*: output* 1
else output 0*.*

Additionally, $\mathcal{A}_2$ *is not allowed to submit* C_b *to oracle* $\mathtt{Dec}_K(\cdot)$*.*

An encryption scheme $\Pi = (\mathtt{Gen}, \mathtt{Enc}, \mathtt{Dec})$ *is said to achieve indistinguishability under adaptive chosen-chiphertext attack (IND-CCA2) if for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, there exists a negligible function such that*

$$\Pr\left[\mathsf{Ind\text{-}CCA2}_{\Pi,\mathcal{A}}(k) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(k), \tag{2.8}$$

where the probabilities are taken over the randomness in $\mathcal{A}$*, the internal coin toss of the* Gen *and* Enc *algorithm and over the choice of b which is sampled uniformly from* $\{0,1\}$*.*

While the adversary had only oracle access to the encryption function in $\mathsf{Ind}_{\mathsf{CPA}}$, in the $\mathsf{Ind}_{\mathsf{CCA2}}$ experiment the adversary is provided with oracle access to encryption and decryption. To exclude the trivial case of asking the oracle if M_0 or M_1 has been encrypted, the adversary is not allowed to submit C_b to the decryption oracle. Since the adversary is allowed to use both encryption and decryption as black box, the IND-CCA2 security model is more powerful than the IND-CPA one. While the used attack setting seems to be more of an academic interest at first glance, a practical attack against an older version of Secure Sockets Layer (SSL) using the RSA cryptosystem showed its practical relevance [Ble98].

For completeness, it has to be mentioned that indistinguishability under non-adaptive chosen-chiphertext attack (IND-CCA1) is the second security definition for chosen ciphertext attacks known in literature. The definitions of IND-CCA1 and IND-CCA2 are identical, except that in IND-CCA1 the adversary is provided with oracle decryption access only until the challenge plaintexts are chosen. Consequently, IND-CCA1 is weaker than IND-CCA2.

As final note and to close the introduction to cryptography, both IND-CCA1 and IND-CCA2 imply IND-CPA. Therefore, the previous discussed need of a randomised encryption clearly also applies to these stronger security models.

2.6. Hash Functions

In the following, the basics of hash functions are discussed following the works of Preneel and Menezes et al. [Pre93; MOV96].

A hash function maps an input of arbitrary, possible very long length to a value of fixed, small length output. The output of a hash function is known as message digest, digest, hash value or simply hash. Before going into more details, it has to be mentioned that hash functions are not only useful in a cryptographic context, but are often used for addressing data, for example in hash tables.

Definition 2.18 (Hash Function)**.** *A n-bit hash function* h *takes a binary input message of arbitrary length and maps it to a hash value of n bits, i.e.,*

$$\mathsf{h} : \{0,1\}^* \longrightarrow \{0,1\}^n. \tag{2.9}$$

Additionally, the hash value $y = \mathsf{h}(x)$ *has to be efficiently computable for all* $x \in \{0,1\}^*$.

In typical applications, the size of the output (hash value) is much smaller than the size of the input (message). Therefore, it is possible that two input messages result in the same hash value. Obviously, the probability of such a collision decreases with increasing the size of the hash value. Since finding a collision could have significant consequences regarding security, cryptographic hash functions need to fulfil additional requirements to secure the integrity of data.

For a cryptographic hash function h, three basic security properties exist:

Preimage Resistance: Given a hash value y, it is computationally infeasible to find an input message x, such that $\mathsf{h}(x) = y$. Functions fulfilling this property are known as one-way functions.

Second Preimage Resistance: Given an input message x and the corresponding hash value $\mathsf{h}(x)$, it is computational infeasible to find a second input message x' with $x' \neq x$, such that $\mathsf{h}(x') = \mathsf{h}(x)$. This property is also known as weak collision resistance in literature.

Collision Resistance: It is computational infeasible to find two distinct input messages x, x', such that $\mathsf{h}(x') = \mathsf{h}(x)$. In contrary to the second preimage resistance, both messages can be freely chosen. This property is also known as strong collision resistance in literature.

2.7. Bloom Filter

The following definition and construction method for Bloom filters is based upon the original work in [Blo70] and their usage in the secure index scheme presented in [Goh03].

A Bloom filter is a probabilistic data structure designed to store a set of elements in way that later testing if a specific element is part of the set can be done efficiently. It is represented by an array of m bits acting as the encoding of a set of messages or input elements. As important feature of the underlying array, it is required that all bits are individually addressable. For encoding a message M, r independent hash functions $\mathsf{h}_1, \ldots, \mathsf{h}_r$ are used. Each of these hash function h_i maps an arbitrary input message to one of the m array bits respectively array positions, i.e., $\mathsf{h}_i = \{0,1\}^* \to [1,m]$ for $i \in [1,r]$. As a matter of course, the mapping of messages to array positions should be uniformly random.

Construction

At first, all m bits of the Bloom filter are initialised to 0. For each message M, the corresponding array positions are computed, i.e., $\mathsf{h}_1(M), \ldots, \mathsf{h}_r(M)$. Afterwards, for all computed positions the respectively bits in the Bloom filter are set to 1. To provide an example, if $\mathsf{h}_1(M)$ outputs value x, then the x^{th} bit in the Bloom filter is set to 1.

Membership Test

To test whether a message M is part of the set $\mathcal{M}$ encoded in the Bloom filter, once again the set of corresponding array positions has to be computed, i.e., $\mathsf{h}_1(M),\ldots,\mathsf{h}_r(M)$. Afterwards, all bits at the computed array positions are tested whether they are set or not. If at least one of the relevant bits is not set, the message M is certainly not part of the encoded set $\mathcal{M}$. If all relevant bits are set to 1, M is per definition part of $\mathcal{M}$. Recalling that during the construction process a bit can be set by different inputs, it is obvious that it can not be distinguished whether a bit was set by a specific message in retrospect. Consequently, even if all relevant bits indicated by the hash values of a message M are set, it could be the case that M was not part of the set, i.e., that a collision happened. Therefore, there exists a probability that false positives occur. In practice, the probability of false positives can be reduced by suitable choices for r and m appropriate for the concrete usage scenario.

Collisions

Depending on the values of r and m, it is more or less likely that an array position will be set to 1 multiple times. Such a collision can occur between two or more input messages, but also if only one message is inserted into the Bloom filter since the output of two hash functions using the same input message could be identical as well, i.e., $\mathsf{h}_i(M) = \mathsf{h}_j(M)$ for $i, j \in [1,r]$. While a collision with just one input message seems unlikely at first glance, it is not that improbable if the array's length should not be sufficient larger than the number of hash functions r. Regardless of the reason for multiple position occurrences while inserting messages, each bit is set to 1 at first need and will neither be further increased nor be reverted to 0.

False Positives

According to [MU05], the probability of a false positive f is given as

$$(1-\mathrm{e}^{-\frac{rn}{m}})^r, \tag{2.10}$$

where r defines the number of hash functions, n the number of inserted elements and m the size of the array. For instantiating a Bloom filter in practice, it is usable to define a desired false positive rate f_p and calculate the other parameters accordingly, i.e., $r = -\log_2(f_p)$ and $m = nr/\ln 2$ [Goh03].

Updates

Due to the used encoding, it is not possible to reconstruct the messages contained in the Bloom filter. Furthermore, the number of encoded messages is hidden in the Bloom filter. While adding new messages is possible at any time, deleting messages is not possible. The reasoning is the same as in the explanation why false positives exist: Let $x = \mathsf{h}_1(M)$ be array position x that was set by message M that shall be deleted now. The bit x could have been set by a second input message M'. Therefore, it can not be removed by setting it to 0 because the encoding of M' would be affected too. Consequently, only the bits that where set by M exclusively have to be removed. Since it is not possible to find out which bits were exclusive set by M, it is not possible to delete messages out of an Bloom filter without storing additional information.

Chapter 3

Searchable Encryption

This chapter provides a comprehensive survey about searchable encryption. At first, this chapter motivates the usability of searchable encryption and provides a categorisation of different types of searchable encryption schemes. Then, security models used to investigate the security guarantees of searchable encryption schemes are presented.

3.1. Introduction

The increasing interest and need for searchable encryption is motivated by the rising popularity and usage of cloud computing, especially cloud storage services. Considering the prevalent scenario of using a cloud storage as private digital data safe to backup files of arbitrary types, it is evident that the cloud provider has full access to all data. As a consequence of the vendor's capabilities, it would be possible to process and disclose information without the user's permission. Even if the cloud provider would be fully trustworthy, there is still a possibility that the outsourced data is revealed as a result of technical issues or attacks. To protect the privacy of the outsourced data, it has to be stored in encrypted form.

As usual, it is possible to increase security at the cost of usability and functionality. Considering local data storages, providing searching capabilities is one of the key requirements. Due to the need that the outsourced data has to be encrypted, all well known search algorithms for plaintexts are not suitable in the cloud storage scenario. Obviously, the naive approach of downloading all data, decrypting it locally and executing a local search on the decrypted data becomes impractical with higher amount of data very soon. In an ideal solution, the server would search all documents without learning anything about the search and stored data and would return only the relevant documents.

There are different ways how search on encrypted data can be performed[1]: If it is known that the amount of data is very small or if performance should be irrelevant for any other reason, Oblivious Random Access Machines (ORAMs) provide the most secure solution since the server does not learn anything about the access pattern and the type of operation [GO96]. Even though ORAMs are asymptotically efficient, their practical performance is not efficient due to the large constants hidden in the big-O notation [Goh03; CM05; Nav15].

[1] How to Search on Encrypted Data: Introduction (Part 1). URL: http://outsourcedbits.org/2013/10/06/how-to-search-on-encrypted-data-introduction-part-1/ (Accessed 10. December 2016)

A slightly more practical solution in terms of performance is known as Functional Encryption (FE). Without going into too much details, FE defines a generalisation for existing asymmetric encryption like Identity-Based Encryption (IBE) and Attribute-Based Encryption (ABE) that can be used to build searchable encryption schemes [BSW11]. Because of the public key nature of the underlying cryptographic functions and the general usability of these primitives, FE is not as efficient as specific searchable encryption solutions.

In search for a fast solution, using Property-Preserving Encryptions (PPEs), which can be realised with deterministic encryption [BBO07], appears to be an option. While PPE achieves sub-linear server-side search performance, it is not as secure as ORAM based solutions because the stored documents and performed searches leak some information a malicious server could exploit. To provide some leakage examples, the server learns if documents contain the same words, the frequency in which words appear and if searches are repeated [2].

For practical usage, specific searchable encryption schemes – in particular symmetric solutions – provide a reasonable trade-off between security and performance. While state-of-the-art Searchable Symmetric Encryption (SSE) schemes achieve sub-linear server-side search performance like PPE solutions, they provide strong security guarantees that are considered secure even in an academic context and seem hence suitable for practical usage. Obviously, the concrete performance and the supported security model depend on the specific searchable encryption scheme.

To sum it up, there exist various different approaches how search on encrypted data can be performed. While most of them are either insecure or do not achieve adequate performance for practical applications, modern SSE schemes provide strong security and usable performance. The following section provides a general overview of the different types of searchable encryption schemes and their applications.

3.2. Categorisation

There are several classifiers that can be used to categorise searchable encryption schemes. As an outlook, it can be distinguished between symmetric and asymmetric schemes, single-user and multi-user schemes and if searches are performed on the data or on some provided metadata often referred to as index. If an index is used, one can differentiate between schemes using forward and inverted indices. Furthermore, schemes can be classified as static and dynamic depending whether updates are possible. Considering the search process, it can be distinguished whether parallel search queries are supported or if the search has to be performed in a sequential way. Finally, the types of supported queries and the degree of interactivity during the search process are another criteria for categorising searchable encryption schemes. In the following, these categories are discussed in more detail.

Symmetric versus Asymmetric Searchable Encryption. In Searchable Symmetric Encryption (SSE), only the owner of the private key is able to encrypt data and store it in a searchable way, generate encrypted search queries and decrypt the retrieved data. Obviously, all other actions like updating the outsourced files are only feasible in possession of the secret key as well. In an Asymmetric Searchable Encryption scheme, also known as Public-Key Encryption with Keyword Search (PEKS) [Cur+06], everyone in possession of the public key is able to encrypt and upload data to the storage, but only the data owner in possession of the private key is able to generate encrypted search queries and decrypt data. However, PEKS schemes are far less efficient than SSE solutions and it is fair to say that PEKS are more of a

[2] How to Search on Encrypted Data: Deterministic Encryption (Part 2). URL: http://outsourcedbits.org/2013/10/14/how-to-search-on-encrypted-data-deterministic-encryption-part-2/ (Accessed 10. December 2016)

research interest and currently not the best choice for practical applications. This is one of the reasons why this thesis focuses on SSE schemes.

Single-User versus Multi-User. Like in many information systems and software architectures, it can be differentiated whether the construction is of a single-user or multi-user type. In the single-user setting, only the data owner is allowed to encrypt data, outsource it in a searchable form, generate search queries and retrieve information. In the multi-user setting, the data owner is still in control of the data, but it is possible to grant search and other access possibilities to other users. While there exist searchable encryption solutions solely for the multi-user setting [RMÖ15], it is possible to build multi-user schemes out of SSE schemes [Cur+06].

Full-Text versus Index-Based Search. Another distinctive feature for schemes is on which data the search is performed. In a full-text search, each document of an outsourced document collection is represented as a sequence of words. Later on, documents can be searched by testing each word in the document against a keyword for equality or some other condition. In the case of SSE, the document has to be encrypted in a special way to allow equality tests against keywords. Since the server should ideally not have any information about the stored contents, it is likely that the whole document collection has to be searched resulting in a search time proportional to the encrypted document collection size. Quite obvious, the performance of full-text search based approaches does not scale well with increasing amount of data, making this approaches impractical very soon.

In contrast, index-based searches perform the search on an additional metadata file often referred to as index or index file. The index, which is created locally based on the content of a document and uploaded together with the corresponding encrypted documents, should ideally not reveal any information about the document collection. The structure of the index does highly depend on the used SSE scheme and could be based on any data structure, for example on a linked list, look-up table or tree. Considering the performance, index-based searches have a significantly better performance compared to full-text searches. Consequently, research has focused on developing fast and secure index-based SSE schemes rather than schemes for full-text search.

Forward Index versus Inverted Index. While a forward index contains a list of keywords per document, an inverted index stores for every keyword a list of pointers to documents containing the keyword. In the context of searchable encryption and regardless of the index type, the list has to be obfuscated somehow to prevent leaking information about the document content.

Static versus Dynamic Schemes. A scheme is classified as dynamic if it is efficiently possible to perform updates regarding the document content and the associated keywords. If updates are not supported or only the trivial solution of rebuilding the whole index in case of an update can be applied, the scheme is categorized as static. Due to the security requirement that information leakage from the stored indices itself shall be minimized, developing schemes that allow modifying the index is a non-trivial task.

Sequential versus Parallel Search. Depending on the type of the internal data structure used in the index, schemes can be classified whether the search process is parallelisable or if sequential searching is needed. As known from search algorithms for plaintext searches, there exist a lot of different data structures that can be applied. To provide some examples, some possible structures used in conventional searches are arrays, linked lists, graphs, heaps, various types of trees like binary trees, red–black trees

and hash based constructions like hash tables or Bloom filters. While some of the constructions require sequential access (e.g. linked lists) implying a sequential search process, others (e.g. trees) support parallel operations resulting in a parallelisable search process.

Types of Keyword Queries. As another noteworthy distinguisher, schemes can be labelled by their support of different query types. In general, it can be distinguished between single keyword and multiple keyword searches. In both types of queries, the search can be seen as predicate that has to be satisfied by documents. The simplest type of a search query contains an equality predicate testing for occurrence of a single keyword in documents. Additionally, some schemes support more complex predicates such as range queries (e.g. $x > 10$), subset queries (e.g. $x \in \mathcal{X}$) or conjunctive queries (e.g. $query_1 \wedge query_2$). Clearly, intersecting the results of several single keyword queries can emulate a conjunctive query. While schemes supporting single keyword searches are able to reproduce simple functions like conjunctions, a straightforward emulation of complex boolean functions might not be possible. Therefore, if the query gets more complicated, schemes supporting arbitrary boolean queries have to be considered.

Regardless of the query's complexity, in terms of efficiency and security, a scheme supporting particular multiple keyword queries might have some advantages over single keyword schemes. To provide an example in the basic conjunctive query case, it is easy to see that the server learns the intersection of single keyword searches and total number of searched keywords if the outcomes of several single keyword searches are intersected, but it is possible to hide these information in conjunctive queries.

Interactivity. Finally, another distinction can be made by the degree of interactivity between client and server. While in a non-interactive scheme an interaction is defined as a single user query to the server resulting in a single answer, in an interactive scheme a larger number of queries and responses is needed to complete a single interaction.

As motivated by this classification, index-based symmetric schemes provide the best performance and security guarantees among searchable encryption schemes. Therefore, the rest of this thesis focuses on SSE schemes.

3.3. Definitions and Notation

In this section and based on the work in [Cur+06], the notation for modelling SSE schemes and existing security models are introduced. Since this notation is used for all later discussed schemes, it can differ from the notation used in the respective papers. Special definitions solely required for single schemes are not covered here and will be introduced at the time of need.

Definition 3.1 (Dictionary)**.** *A dictionary Δ contains d words $(w_1, \ldots, w_d)$ in lexicographic order, where each word $w_i \in \Delta$ is a binary string.*

A dictionary defines the universe of words that might occur in documents. Given a security parameter k, the total number of words d is bounded by an arbitrary polynomial function $\mathsf{poly}(k)$ and the length of each keyword $w \in \Delta$ is polynomial in k.

Definition 3.2 (Document Collection)**.** *A document collection $\boldsymbol{\mathcal{D}} \subseteq (2^{\Delta})^n$ is defined as a collection of n documents, i.e., $\boldsymbol{\mathcal{D}} = (D_1, \ldots, D_n)$, where 2^{Δ} defines the set of possible documents in respect to the dictionary Δ and both - the number of documents n and the number of words in each document - are bounded by an unspecified polynomial function* $\mathsf{poly}(k)$.

The set of distinct words in the document collection is indicated as $\Delta' = \delta(\boldsymbol{\mathcal{D}})$ with $\Delta' \subseteq \Delta$. In order to label documents, each document $D \in \boldsymbol{\mathcal{D}}$ has a unique identifier accessible by $\mathsf{id}(D)$. The lexicographic ordered set of document identifiers of all documents in a document collection $\boldsymbol{\mathcal{D}}$ containing keyword w is denoted by $\boldsymbol{\mathcal{D}}(w)$.

Index-Based SSE Scheme

For a better understanding of the upcoming security requirements and later introduced security models, an abstract index-based SSE scheme is presented below. Since the definitions of index-based SSE schemes vary in literature [Goh03; Cur+06; KP13; Cas+13], the successive definition can be seen as unified model that covers all later presented schemes.

Definition 3.3 (Index-Based SSE Scheme)**.** *An index-based SSE scheme is a tuple of algorithm* $(\texttt{KeyGen}, \texttt{BuildIndex}, \texttt{Trapdoor}, \texttt{Search})$ *such that:*

$\texttt{KeyGen}(1^k)$*: This probabilistic algorithm takes a security parameter k as input and outputs a secret key K.*

$\texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}})$*: This possibly probabilistic algorithm takes a secret key K and a set of documents $\boldsymbol{\mathcal{D}}$ as inputs and outputs an encrypted index I.*

$\texttt{Trapdoor}(K, w)$*: This deterministic algorithm returns the trapdoor τ_w for a given secret key K and keyword w.*

$\texttt{Search}(I, \tau_w)$*: This deterministic algorithm performs the search for keyword w in a document collection $\boldsymbol{\mathcal{D}}$ specified by an encrypted index I given trapdoor τ_w and outputs a list of found document identifiers.*

Additionally, for all security parameters $k \in \mathbb{N}$, for all keys K generated by $\texttt{KeyGen}(1^k)$*, for all indices I computed by* $\texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}})$ *for all possible document collections, the property*

$$\texttt{Search}(I, \texttt{Trapdoor}(K, w)) = \boldsymbol{\mathcal{D}}(w)$$

has to hold for all keywords $w \in \Delta$.

Additionally, the implications of using a dictionary and the way document encryption is handled have to be explained.

Dictionary Usage: Depending on the specific scheme, a dictionary might be used by the `BuildIndex` algorithm. By using a dictionary, the universe of keywords can be restricted to essential ones to avoid overspecialised keywords and to prevent using various synonyms as keywords.

Considering dictionary updates, it is possible to delete keywords without updating the index, but keyword positions can not be reused. If keywords shall be added, the initially defined upper bound of the total number of keywords can not be exceeded without rebuilding the index. Consequently, it is not possible to add an unlimited amount of keywords without issues, but it would be possible to find some workarounds for this problem. For example, one could delay the update if the index does not contain the new keywords by assuming that all exceeding keywords are not part of the index. While such simple update approaches are imaginable, the information leakage that specific keywords are not included would be introduced immediately. From a security perspective, updating the index would be the only reasonable way to deal with dictionaries extending the initial upper bound.

Document Encryption: Obviously, encrypting and decrypting the document collection has to be part of the searchable encryption solution as well. Since standard cryptographic mechanism can be used to secure the confidentiality of the outsourced document collection, it is not a major concern for index-based SSE schemes. Consequently, all well known and established standard encryption algorithms that are proven to be secure can be used. Hence, information leakage solely from the ciphertexts does not have to be covered in SSE security notions.

Depending on the specific SSE scheme, the document encryption is either done at first outside the SSE scheme, or during the `BuildIndex` algorithm. In both cases, it results in an encrypted index and an encrypted document collection after the `BuildIndex` algorithm has been computed.

For a better understanding of the defined algorithms, the typical use case of applying the SSE algorithm is discussed.

Key Generation: To generate a key K, the `KeyGen` algorithm is executed client-side by the user.

Index Computation: To build the encrypted index I for a set of documents $\mathcal{D}$, the `BuildIndex` is run client-side by the user. Depending on the type of scheme, I can either be a forward or inverted index. Additionally, the documents are encrypted using a standard encryption scheme. Afterwards, both I and the encryption of $\mathcal{D}$ can be stored on a server.

Search: To search for a keyword w, the corresponding trapdoor τ_w has to be computed client-side using the `Trapdoor` algorithm and key K. Afterwards, given trapdoor τ_w, the server runs the `Search` algorithm to search for documents using the outsourced index I. To be precise, the `Search` algorithm computes a sequence of typically lexicographically ordered document identifiers that contain a keyword w hidden in trapdoor τ_w using the encrypted index I representing the set of documents $\mathcal{D}$. Given the found document identifiers, the user is able to retrieve and decrypt the corresponding documents.

For the subsequent discussion of security requirements and models, it is important to stress that all computations, except for the `Search` algorithm, are run on the client-side by the user.

3.4. Security Requirements and Models

Considering the security requirements for searching on encrypted data, the question which kind of information a server is allowed to learn arises. In general, possible information leakage can occur on stored data or from the interactions between the server and the user's device.

Stored Data: Regarding all stored data and data received by the server, information could leak from the encrypted files, the index and the trapdoors. As examples, some of the possible information leakages could be the number of documents, the document sizes, the document identifiers, the number of keywords in a document, the total number of keywords, the number of keywords documents share and other similarity disclosures of document sets.

Search Process: The more complex part of information leakage are the implications of the search process and other interactions between server and user. Finding security models that deal with all kind of information disclosures and developing schemes that satisfy the required security grantees has been of great interest in research over the past decade and is still ongoing. The currently most common requirement was defined by [Cur+06] stating that nothing should be leaked except the search and access pattern.

Before defining these requirements and terms formally, the evolution of security models is summarised to understand the goals of state-of-the-art solutions.

3.4.1. History of Security Models

According to [CM05; Cur+06], the full-text scheme presented in [SWP00] was the first encryption scheme targeted for searchable encryption achieving indistinguishability under chosen-plaintext attacks (IND-CPA) [KP13]. As stated by Curtmola et al., the fundamental issue with IND-CPA for SSE is that it only attests that the underlying construction is a secure encryption scheme, but it does not prove that it is a secure searchable encryption scheme [Cur+06]. Considering the full-text scheme published in [SWP00], not only the ciphertexts have to be protected, but also the trapdoors. In the case of an index-based SSE scheme, the index and all information involved in the search process have to be taken into account. Consequently, the notation of IND-CPA is not suitable for SSE schemes.

In [Goh03], Goh defined semantic security against an adaptively chosen keyword attack referred to as IND1-CKA. In a chosen keyword attack, an adversary can choose keywords and documents and the client computes the corresponding indices and trapdoors. Informally, the game based IND1-CKA definition states that an adversary should not be able to gain knowledge about the document content solely from the index. The idea is if I is indistinguishable, i.e., the probability that an adversary is able to deduce whether D_0 or D_1 with $D_0 \neq D_1$ and $|D_0| = |D_1|$ has been encrypted is negligible, then deducing at least one word that appears only in D_0 or D_1 from I is not possible. However, the information leakage from search query results and the leakage from encrypted documents are not covered by IND1-CKA.

Later on, Goh introduced a stronger version known as IND2-CKA. In contrast to IND1-CKA, the documents D_0, D_1 can be of arbitrary, possibly distinct size as long as there exists at least one word that appears only in D_0 or D_1. Aside from the unrestricted choice of documents, the rest of the game based definition of IND2-CKA is equivalent to IND1-CKA. As a consequence, the document size is also protected in IND2-CKA.

Between the publication of IND1-CKA and IND2-CKA, Chang et al. proposed a security model known as IND-CKA [CM05]. Like in IND2-CKA, the documents can be of arbitrary length, but this time they are not restricted in any way. The simulation based definition states that real search interactions do not leak more information than the ideal case where no trapdoors and indices are used. Consequently, no information is leaked from trapdoors and indices making this security model stronger than IND2-CKA.

In the seminal work of Curtmola et al. [Cur+06], all of the above presented security models have been shown to be incorrect or insecure. To start with the IND1-CKA and IND2-CKA models from [Goh03], both deal with the problem that they do not take trapdoors into consideration. Since trapdoor security is not covered, schemes leaking information from trapdoors could still be considered IND1-CKA/IND2-CKA secure. Consequently, both models are insufficient for measuring searchable encryption security. In search for a simple solution, introducing an additional formal requirement stating that no information should be leaked by the trapdoors could be thought of. However, Curtmola et al. claim that simple extensions can not be done trivially and provide an instantiation that fulfils all requirements but results in an insecure scheme [Cur+06].

In the IND-CKA model from [CM05], trapdoor security is also considered. As shown by Curtmola et al., there are issues with the formalisation of IND-CKA as the definition can be satisfied by any arbitrary, possibly insecure SSE scheme [Cur+06]. Therefore, none of the so far discussed security models seem appropriate for describing the desired security of searchable encryption.

The work of Curtmola et al. [Cur+06] not only demonstrated the problems with previous security definitions, but also introduced two new security models that can be seen as de facto standard for describing security for searchable encryption and are widely adapted for different kinds of SSE constructions.

In order to have all components for describing the new security models from [Cur+06] and to have a basic vocabulary for describing and analysing the whole search process of all later presented SSE schemes, some widely used terms have to be formalised.

3.4.2. Search Process Vocabulary

The following definitions are based on [Cur+06].

Definition 3.4 (History)**.** *A history $\mathcal{H} = (\boldsymbol{\mathcal{D}}, \hat{w})$ is a tuple containing a document collection $\boldsymbol{\mathcal{D}}$ and a list of q keywords $\hat{w} = (w_1, \ldots, w_q)$.*

Due to the number of keywords, such a history is also called q-query history. Considering a search process from the user's perspective, a set of documents has to be tested for occurrence of words. It is easy to see that searching for q keywords $\hat{w}$ in a set of documents $\boldsymbol{\mathcal{D}}$ matches the given notation of a q-query history $\mathcal{H}$. Obviously, the server shall not learn the keywords a user wants to search for.

Definition 3.5 (Access Pattern)**.** *Given a q-query history $\mathcal{H} = (\boldsymbol{\mathcal{D}}, \hat{w})$, the access pattern $\alpha(\mathcal{H})$ is defined as $\alpha(\mathcal{H}) = (\boldsymbol{\mathcal{D}}(w_1), \ldots, \boldsymbol{\mathcal{D}}(w_q))$.*

While the history defines which keywords and documents have to be searched, the access pattern defines the result of the search request, i.e., the documents that contain the searched keywords. By reason that the server should not see the keywords, the access pattern reveals information about the outcomes of queries without disclosing the keywords. To be precise, a server learns for each query consisting of q unknown keywords the outcome of the search, i.e., a set of document identifiers corresponding to the stored encrypted documents. Furthermore, the server learns correlations between multiple query results. To provide some simple examples, if two single keyword queries for keywords w_1, w_2 return $\boldsymbol{\mathcal{D}}(w_1)$ and $\boldsymbol{\mathcal{D}}(w_2)$, it can be seen if two unknown keywords occur in the same document. Assuming two single keywords queries resulting in $|\boldsymbol{\mathcal{D}}(w_1)| = 1$ and $|\boldsymbol{\mathcal{D}}(w_2)| > 1$ with $\boldsymbol{\mathcal{D}}(w_1) \subset \boldsymbol{\mathcal{D}}(w_2)$, then keyword w_1 is more restrictive than keyword w_2. While such results seem unlikely at first glance, they could happen easily as the simple example of searching for a person in the first query and searching for persons of a specific gender in the second one shows. Obviously, with larger amounts of queries, the server learns more and more relations between query results. As a final thought, the access pattern aims only for correlations of search results, but not for the search queries itself.

Definition 3.6 (Search Pattern)**.** *Given a q-query history $\mathcal{H} = (\boldsymbol{\mathcal{D}}, \hat{w})$, the search pattern $\sigma(\mathcal{H})$ is defined as symmetric binary $q \times q$ matrix such that for $1 \leq i, j \leq q$:*

$$\sigma[i, j] = \begin{cases} 1 & \text{if } w_i = w_j \\ 0 & \text{else} \end{cases}. \tag{3.1}$$

While the access pattern covers information implied by search outcomes, the search pattern indicates the correlation between queries, in particular of the involved keywords. To be precise, the search pattern reveals whether identical keywords were used in different queries. In general, the search pattern is leaked if the server is able to determine multiple occurrences of keywords. Since the majority of SSE schemes use deterministic trapdoors, it is easy to see that the search pattern can be obtained without further effort. While the server might learn correlations between used keywords, it has to be emphasised that the actual keywords are protected because of their encoding in trapdoors during the search request. However, with increasing amount of searches resulting in a more valuable search pattern, the probability that the server can calculate partial information about the actual keywords by applying statistical attacks increases.

Definition 3.7 (Trace)**.** *Given a q-query history $\mathcal{H} = (\boldsymbol{\mathcal{D}}, \hat{w})$ over a document collection of length $n = |\boldsymbol{\mathcal{D}}|$, the trace $\tau(\mathcal{H})$ of history $\mathcal{H}$ is defined as $\tau(\mathcal{H}) = (\mathsf{id}(D_1), \ldots, \mathsf{id}(D_n), |D_1|, \ldots, |D_n|, \alpha(\mathcal{H}), \sigma(\mathcal{H}))$.*

The trace consists of the document identifiers and lengths of all documents, the access and the search pattern. As already mentioned, the currently most common informal requirement for SSE security states

that nothing should be leaked except the search result and the search pattern. In practical scenarios, the server can most likely learn some metadata about the stored encrypted documents. If no padding or other techniques are used, this leaked metadata consists of the length and identifier of each document. It is easy to see that the given definition of a trace contains all the mentioned data that is sacrificed for reasonable performance and security. Hence, the trace covers all information a server is allowed to learn. As a final note, from a security perspective, it is necessary that at least one second history $\mathcal{H}' \neq \mathcal{H}$ with the same trace exists, i.e., $\tau(\mathcal{H}) = \tau(\mathcal{H}')$. If such a history $\mathcal{H}'$ can be found in polynomial time, it is called a non-singular history.

Definition 3.8 (View)**.** *Given a document collection $\boldsymbol{\mathcal{D}}$ of length $n = |\boldsymbol{\mathcal{D}}|$ with corresponding index I and ciphertexts $\mathcal{C} = (C_1, \ldots, C_n)$ where C_i is the encryption of document D_i for $i \in [1, n]$, the view $\nu = (I, \mathcal{C}, \mathcal{T})$ consists of the index I, the set of ciphertexts $\mathcal{C}$ and a set of trapdoors $\mathcal{T}$.*

Intuitively, the notation of a view contains all data a malicious server or any other adversary could access. Consequently, information leakage based solely on the view has to be prevented.

In the following, formal definitions of the state-of-the-art security models are reviewed.

3.4.3. IND-CKA1

In this section and based on the work in [Cur+06], the non-adaptive security models achieving non-adaptive indistinguishability for Searchable Symmetric Encryption (IND-CKA1) are discussed. For this purpose, let $st_{\mathcal{A}}$ be the adversary's state information, i.e., a string storing the state of adversary $\mathcal{A}$.

IND-CKA1 Semantic Security

Definition 3.9 (IND-CKA1 Semantic Security)**.** *Let SSE $= (\texttt{KeyGen}, \texttt{BuildIndex}, \texttt{Trapdoor}, \texttt{Search})$ be an index-based SSE scheme, $\mathcal{A}$ be an adversary, $k \in \mathbb{N}$ be a security parameter and $\mathcal{S}$ be a simulator. The real world experiment $\mathsf{Real}_{SSE,\mathcal{A}}(k)$ is defined as*

$\mathsf{Real}_{SSE,\mathcal{A}}(k)$
- $K \leftarrow \texttt{KeyGen}(1^k)$
- $(\mathcal{H}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^k)$
- *parse* $\mathcal{H}$ *as* $(\boldsymbol{\mathcal{D}}, \hat{w})$
- $(I, \mathcal{C}) \leftarrow \texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}})$
- *for* $1 \leq i \leq q$:
  - $\tau_i \leftarrow \texttt{Trapdoor}(K, w_i)$
- *let* $\mathcal{T} = (\tau_1, \ldots, \tau_q)$ *and* $\nu = (I, \mathcal{C}, \mathcal{T})$
- *output* ν *and* $st_{\mathcal{A}}$

and the ideal world experiment $\mathsf{Ideal}_{SSE,\mathcal{A},\mathcal{S}}(k)$ is defined as

$\mathsf{Ideal}_{SSE,\mathcal{A},\mathcal{S}}(k)$
- $(\mathcal{H}, st_{\mathcal{A}}) \leftarrow \mathcal{A}(1^k)$
- $\nu \leftarrow \mathcal{S}(\tau(\mathcal{H}))$
- *output* ν *and* $st_{\mathcal{A}}$.

A SSE scheme SSE is said to be IND-CKA1 semantically secure if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, there exists a polynomial time simulator $\mathcal{S}$ such that for all PPT distinguishers D*,*

$$\begin{aligned} &\Big| \Pr\left[\mathtt{D}(\nu, st_{\mathcal{A}}) = 1 : (\nu, st_{\mathcal{A}}) \leftarrow \mathsf{Real}_{SSE,\mathcal{A}}(k)\right] - \\ &\quad \Pr\left[\mathtt{D}(\nu, st_{\mathcal{A}}) = 1 : (\nu, st_{\mathcal{A}}) \leftarrow \mathsf{Ideal}_{SSE,\mathcal{A},\mathcal{S}}(k)\right] \Big| \leq \mathsf{negl}(k), \end{aligned} \tag{3.2}$$

where the probabilities are taken over randomness of the probabilistic KeyGen *and* BuildIndex *algorithm.*

In the real world Real, the adversary creates a history $\mathcal{H}$ consisting of a document collection $\boldsymbol{\mathcal{D}}$ and a sequence of keywords $\hat{w}$ that the adversary wants to search for. It is important to emphasise that in the present non-adaptive setting, the history has to be generated at once. Hence, an adversary can not access data before generating the history is finished. Additionally, the adversary is allowed to save a string $st_{\mathcal{A}}$ providing arbitrary information about its state.

The rest of the experiment works straightforward: The encrypted index I and the encrypted document collection $\mathcal{C} = (C_1, \ldots, C_n)$ are built using the BuildIndex algorithm. Next, for each keyword $w_i \in \hat{w}$ of the q-query history $\mathcal{H}$, a corresponding trapdoor τ_i is computed utilising the Trapdoor algorithm. Finally, the real world experiment outputs a view $\nu = (I, \mathcal{C}, \mathcal{T})$ where $\mathcal{T}$ is the set of all created trapdoors. Additionally, the adversary's state information $st_{\mathcal{A}}$ is added to the output.

All in all, the Real experiment covers a situation where an adversary is able to use the SSE algorithms to generate a view of an arbitrary history. Even though the adversary is allowed to construct the history, the initial security requirement stating that nothing should be leaked except the search and the access pattern has to be fulfilled.

Applying the general concept of simulation based proofs, in the ideal world the cryptographic primitive that shall be tested is not even used. Therefore, Ideal does not apply any functions of *SSE*. At first the adversary provides the same q-query history $\mathcal{H}$ and the state information $st_{\mathcal{A}}$. Next, the view is generated by the simulator $\mathcal{S}$ receiving the trace $\tau(\mathcal{H})$ of $\mathcal{H}$ and Ideal returns the simulated view ν and $st_{\mathcal{A}}$. Since only the trace of the history was used to generate the output, information that might be leaked from the simulated view can be seen as uncritical.

Revisiting Equation 3.2, a SSE scheme provides IND-CKA1 semantic security if the view computed in the real world experiment can not be distinguished from a simulated view generated in the ideal world experiment with more than negligible advantage. Bearing in mind that the real world view consists of the encrypted index, the encrypted document collection and the trapdoors, it is obvious that all these data do not leak any critical information. To be precise, the only information an adversary can learn is the information included in the trace.

IND-CKA1 Indistinguishability

Definition 3.10 (IND-CKA1 Indistinguishability)**.** *Let SSE* $= (\texttt{KeyGen}, \texttt{BuildIndex}, \texttt{Trapdoor}, \texttt{Search})$ *be an index-based SSE scheme,* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *be an adversary and* $k \in \mathbb{N}$ *be a security parameter. The experiment* $\mathsf{Ind}_{SSE,\mathcal{A}}(k)$ *is defined as*

$\mathsf{Ind}_{SSE,\mathcal{A}}(k)$
- $K \leftarrow \texttt{KeyGen}(1^k)$
- $(st_{\mathcal{A}}, \mathcal{H}_0, \mathcal{H}_1) \leftarrow \mathcal{A}_1(1^k)$
- $b \xleftarrow{\$} \{0,1\}$
- *parse* $\mathcal{H}_b$ *as* $(\boldsymbol{\mathcal{D}}_b, \hat{w}_b)$
- $(I_b, \mathcal{C}_b) \leftarrow \texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}}_b)$
- *for* $1 \leq i \leq q$ *:*
  - $\tau_{b,i} \leftarrow \texttt{Trapdoor}(K, w_{b,i})$
- *let* $\mathcal{T}_b = (\tau_{b,1}, \ldots, \tau_{b,q})$
- $b' \leftarrow \mathcal{A}_2(st_{\mathcal{A}}, I_b, \mathcal{C}_b, \mathcal{T}_b)$
- *if* $b' = b$*: output* 1
- *else output* 0*,*

where the adversary has to choose $\mathcal{H}_0$ *and* $\mathcal{H}_1$ *in a way that the traces are equal, i.e.,* $\tau(\mathcal{H}_0) = \tau(\mathcal{H}_1)$*. A SSE scheme SSE achieves IND-CKA1 indistinguishability if for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*,*

$$\Pr\left[\mathsf{Ind}_{SSE,\mathcal{A}}(k) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(k), \tag{3.3}$$

where the probability is taken over the choice of b which is sampled uniformly from $\{0,1\}$ *and the internal coin tosses of* `KeyGen` *and* `BuildIndex`*.*

In the Ind experiment, the adversary creates two histories $\mathcal{H}_0$ and $\mathcal{H}_1$ with identical traces, i.e., $\tau(\mathcal{H}_0) = \tau(\mathcal{H}_1)$. Consequently, the histories are non-singular. Afterwards, a bit b is sampled uniformly from $\{0,1\}$ determining whether history $\mathcal{H}_0$ or $\mathcal{H}_1$ will be used in the upcoming computations. Depending on b, the index I_b, the encrypted document collection $\mathcal{C}_b$ and a set of trapdoors $\mathcal{T}_b = (\tau_{b,1}, \ldots, \tau_{b,q})$ are calculated as usual, i.e as in the Real experiment of IND-CKA1 semantic security.

Provided with I_b ,$\mathcal{C}_b$, $\mathcal{T}_b$ and $st_{\mathcal{A}}$, the adversary has to solve the challenge of determining whether $\mathcal{H}_0$ or $\mathcal{H}_1$ has been used. Consequently, Ind models a setting where an adversary $\mathcal{A}$ is able to generate inputs and has to determine which one was used by the SSE scheme.

As formally defined in Equation 3.3 and following the principles of game based definitions (see Section 2.5.5 for the similar definition of IND-CPA), the index-based SSE scheme used in Ind achieves non-adaptive indistinguishability if for all PPT adversaries, the probability that the adversary can determine correctly if $\mathcal{H}_0$ or $\mathcal{H}_1$ has been processed is at most negligible better than guessing.

3.4.4. IND-CKA2

In this section and based on the work in [Cur+06], the adaptive security models known as adaptive indistinguishability for Searchable Symmetric Encryption (IND-CKA2) are discussed.

IND-CKA2 Semantic Security

Definition 3.11 (IND-CKA2 Semantic Security)**.** *Let* $SSE = (\texttt{KeyGen}, \texttt{BuildIndex}, \texttt{Trapdoor}, \texttt{Search})$ *be an index-based SSE scheme,* $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$ *with* $q \in \mathbb{N}$ *be an adversary,* $k \in \mathbb{N}$ *be a security parameter and* $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ *be a simulator. The real world experiment* $\mathsf{Real'}_{SSE,\mathcal{A}}(k)$ *is defined as*

$\mathsf{Real'}_{SSE,\mathcal{A}}(k)$
- $K \leftarrow \texttt{KeyGen}(1^k)$
- $(\boldsymbol{\mathcal{D}}, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^k)$
- $(I, \mathcal{C}) \leftarrow \texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}})$
- $(w_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, I, \mathcal{C})$
- $\tau_1 \leftarrow \texttt{Trapdoor}(K, w_1)$
- *for* $2 \leq i \leq q$:
  - $(w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, I, \mathcal{C}, \tau_1, \ldots, \tau_{i-1})$
  - $\tau_i \leftarrow \texttt{Trapdoor}(K, w_i)$
- *let* $\mathcal{T} = (\tau_1, \ldots, \tau_q)$ *and* $\nu = (I, \mathcal{C}, \mathcal{T})$
- *output* ν *and* $st_{\mathcal{A}}$

and the ideal world experiment $\mathsf{Ideal'}_{SSE,\mathcal{A},\mathcal{S}}(k)$ *is defined as*

$\mathsf{Ideal'}_{SSE,\mathcal{A},\mathcal{S}}(k)$
- $(\boldsymbol{\mathcal{D}}, st_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^k)$
- $(I, \mathcal{C}, st_{\mathcal{S}}) \leftarrow \mathcal{S}_0(\tau(\boldsymbol{\mathcal{D}}))$
- $(w_1, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, I, \mathcal{C})$
- $(\tau_1, st_{\mathcal{S}}) \leftarrow \mathcal{S}_1(st_{\mathcal{S}}, \tau(\boldsymbol{\mathcal{D}}, w_1))$
- *for* $2 \leq i \leq q$:
  - $(w_i, st_{\mathcal{A}}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, I, \mathcal{C}, t_1, \ldots, t_{i-1})$
  - $(\tau_i, st_{\mathcal{S}}) \leftarrow \mathcal{S}_i(st_{\mathcal{S}}, \tau(\boldsymbol{\mathcal{D}}, w_1, \ldots, w_i))$
- *let* $\mathcal{T} = (\tau_1, \ldots, \tau_q)$ *and* $\nu = (I, \mathcal{C}, \mathcal{T})$
- *output* ν *and* $st_{\mathcal{A}}$.

A SSE scheme SSE is said to be IND-CKA2 semantically secure if for all PPT adversaries $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$ *with q being polynomially bounded, there exists a polynomial time simulator* $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ *such that for all PPT distinguishers* D*,*

$$\left| \Pr\left[\mathtt{D}(\nu, st_{\mathcal{A}}) = 1 : (\nu, st_{\mathcal{A}}) \leftarrow \mathsf{Real'}_{SSE,\mathcal{A}}(k)\right] - \Pr\left[\mathtt{D}(\nu, st_{\mathcal{A}}) = 1 : (\nu, st_{\mathcal{A}}) \leftarrow \mathsf{Ideal'}_{SSE,\mathcal{A},\mathcal{S}}(k)\right] \right| \leq \mathsf{negl}(k), \tag{3.4}$$

where the probabilities are taken over the internal coin tosses in KeyGen *and* BuildIndex*.*

The upcoming IND-CKA2 explanation focuses on the differences of the experiment to the IND-CKA1 experiment.

Considering Real', it can be seen that the history is constructed adaptively instead of at once. To be precise, the index I and the encrypted document collection $\mathcal{C}$ are built as usual, but the set of trapdoors $\mathcal{T}$ is generated adaptively. For this task, the adversary receives $st_{\mathcal{A}}, I, \mathcal{C}$ and all so far computed trapdoors. Utilising this data, the adversary returns the next keyword, a new $st_{\mathcal{A}}$ and the corresponding trapdoor is computed. This process is repeated until all q trapdoors have been produced. This adaptive generation corresponds to a practical situation where a document collection is stored at a server and searches are performed one after the other. Regardless of the current number of searches, the initial security requirement stating that nothing should be leaked except the search and the access pattern has to be fulfilled.

In Ideal', the general procedure for creating the view ν is the same as in Real', but tasks where the SSE scheme has to be used are simulated as usual. Following the same reasoning as in IND-CKA1, information that might be leaked from the simulated view can be classified as uncritical.

IND-CKA2 Indistinguishability

Definition 3.12 (IND-CKA2 Indistinguishability)**.** *Let* $SSE = (\texttt{KeyGen}, \texttt{BuildIndex}, \texttt{Trapdoor}, \texttt{Search})$ *be an index-based SSE scheme,* $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_{q+1})$ *with* $q \in \mathbb{N}$ *be an adversary and* $k \in \mathbb{N}$ *be a security parameter. The game based experiment* $\mathsf{Ind'}_{SSE,\mathcal{A}}(k)$ *is defined as*

$\mathsf{Ind'}_{SSE,\mathcal{A}}(k)$
- $K \leftarrow \texttt{KeyGen}(1^k)$
- $(st_{\mathcal{A}}, \boldsymbol{\mathcal{D}}_0, \boldsymbol{\mathcal{D}}_1) \leftarrow \mathcal{A}_0(1^k)$
- $b \xleftarrow{\$} \{0,1\}$
- $(I_b, \mathcal{C}_b) \leftarrow \texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}}_b)$
- $(st_{\mathcal{A}}, w_{0,1}, w_{1,1}) \leftarrow \mathcal{A}_1(st_{\mathcal{A}}, I_b, \mathcal{C}_b)$
- $\tau_{b,1} \leftarrow \texttt{Trapdoor}(K, w_{b,1})$
- *for* $2 \leq i \leq q$:
  - $(st_{\mathcal{A}}, w_{0,i}, w_{1,i}) \leftarrow \mathcal{A}_i(st_{\mathcal{A}}, I_b, \mathcal{C}_b, \tau_{b,1}, \ldots, \tau_{b,i-1})$
  - $\tau_{b,1} \leftarrow \texttt{Trapdoor}(K, w_{b,i})$
- *let* $\mathcal{T}_b = (\tau_{b,1}, \ldots, \tau_{b,q})$
- $b' \leftarrow \mathcal{A}_{q+1}(st_{\mathcal{A}}, I_b, \mathcal{C}_b, \mathcal{T}_b)$
- *if* $b' = b$*: output* 1
- *else output* 0*,*

where the adversary has to choose $\boldsymbol{\mathcal{D}}_0, \boldsymbol{\mathcal{D}}_1$ *and the keywords in a way that the traces are equal, i.e.,* $\tau(\boldsymbol{\mathcal{D}}_0, w_{0,1}, \ldots, w_{0,q}) = \tau(\boldsymbol{\mathcal{D}}_1, w_{1,1}, \ldots, w_{1,q})$*. A SSE scheme SSE achieves IND-CKA2 indistinguishability if for all PPT adversaries* $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_{q+1})$ *with q being polynomially bounded,*

$$\Pr\left[\mathsf{Ind'}_{SSE,\mathcal{A}}(k) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(k), \tag{3.5}$$

where the probability is taken over the choice of b which is sampled uniformly from $\{0,1\}$ *and the internal coin tosses of* `KeyGen` *and* `BuildIndex`*.*

As one might expect, the definitions of IND-CKA1 and IND-CKA2 indistinguishability are quite similar. Therefore, only the differences of the underlying experiments are clarified in the upcoming paragraph.

While in Ind the non-adaptive history was generated at once, the adversary is allowed to choose an adaptive constructed history in Ind'. This time, it is required that the adaptive generated histories are non-singular to prevent trivial distinguishing. Analysing the adaptive history generation process, it can be seen that the adversary receives $st_{\mathcal{A}}, I_b, C_b$ and all previously constructed trapdoors and has to return two keywords, one for each document collection. As usual, only the keyword matching the random choice of b will be encoded and added to the set of trapdoors. To sum it up, Ind' models a situation where an adversary $\mathcal{A}$ has to determine which of the self created document collections and adaptively generated set of keywords was encoded. Following the reasoning of IND-CKA1, a SSE scheme is IND-CKA2 secure if the distinguishing task can not be done better than simple random guessing.

3.5. Known Attacks on Searchable Symmetric Encryption Schemes

While the security models for describing SSE schemes have been established in the year 2006, the introduced information leakage has not been seen as critical back then and the implications have not been further investigated for a fairly long time and are still not fully understood.

In 2012, the first remarkable attempt to exploit the access pattern leakage was described [IKK12]. In the attack trying to recover the keywords of the queries, the authors assume that the adversary has knowledge about the document collection in form of a fixed set of m keywords and a $m \times m$ matrix M describing keyword co-occurrence probabilities. Due to the access pattern leakage of q distinct trapdoors, the adversary is able to construct a $q \times q$ unknown keyword co-occurrence matrix. The attack uses simulated annealing to find the best match of the observed matrix to a sub matrix in M.

In [Cas+15], an improved keyword recovery attack using similar constraints was presented. Additionally to the co-occurrence matrix M, the adversary has knowledge of the number of documents matching a keyword. At first, all keywords matching a unique number of documents are used to build a known query map. For each remaining query, the set of possible keywords can be build using the initially granted knowledge. Afterwards, the set of possible keywords is reduced until the keywords can be determined by testing if the co-occurrences of the keywords candidates are suitable.

Recently, an active keyword recovery attack, where an adversary is able to add documents of own choice to the document collection, was published, i.e., the adversary sends documents to the client which uses the SSE scheme as intended to perform the update [ZKP16]. In contrast to the previous attacks, the adversary has knowledge of a set of keywords Δ where $|\Delta|$ defines the number of keywords. The attack can be modified to be executed under the same leakages as the previously discussed ones, i.e., partial knowledge of the document collection. In the binary-search attack exploiting the knowledge of Δ, $\log|\Delta|$ files are injected where the i^{th} file contains all keywords whose i^{th} bit is set. After all files have been injected, a search for an encrypted unknown keyword returns a subset of the injected documents. Since the adversary learns for each returned injected file a bit which has to be set in the encoded keyword, the keyword can be recovered. By splitting the keywords into groups and injecting files in a slightly modified way, the number of needed document queries can be further improved.

While no attack without a priori knowledge exists at the time of writing, it is fair to say that the implications of the introduced information leakage are not fully understood so far and will be of great interest in research for exploiting current SSE schemes and constructing improved ones.

Chapter 4

Searchable Encryption Schemes

In this chapter, some of most important Searchable Symmetric Encryption (SSE) schemes are analysed. As already indicated, we focus on symmetric index-based constructions, i.e., the secure index scheme [Goh03], the Privacy Preserving Keyword Searches on Remote Encrypted Data (PPSED) scheme [CM05], the SSE-1 scheme [Cur+06], the keyword red-black (KRB) scheme [KP13] and the Oblivious Cross-Tags (OXT) scheme [Cas+13]. Readers mainly interested in a comparison of all schemes are referred to Chapter 7, especially Table 7.1 which provides a compact comparison of asymptotic efficiency.

4.1. Secure Index Scheme

In this section, the secure index scheme developed by Goh is presented [Goh03].

4.1.1. Idea

The idea of the secure index scheme is to build a forward index for each file in the document collection. The underlying index called Z-IDX is constructed by using pseudorandom functions (PRFs) and Bloom filters. Consequently, the secure index scheme might produce false positive results, for the benefit of a remarkable simple construction with space efficient indices.

As an outlook, the secure index scheme using Z-IDX as index requires $O(1)$ search time per document resulting in a search time linear in the size of the document collection. The space requirements depend on the desired false positive rate and the estimated number of unique words in the document collection, but even with a low false positive rate, the resulting index for medium sized documents needs only a few kilobytes storage space if stored efficiently.

4.1.2. Construction

In typical usage scenarios, one is interested to work with a set of documents rather than with single documents. Since the secure index scheme is of a forward index type, additional steps have to be performed if more than one document shall be stored and searched. In the following, the Z-IDX index construction

building a forward index for document $D_i \in \boldsymbol{\mathcal{D}}$ with $i \in [1,n]$ is provided. Afterwards, the secure index scheme suitable to work with document collections is presented.

Z-IDX

Let k be a security parameter, m be the size of the Bloom filter, $\boldsymbol{\mathcal{D}} = (D_1, \ldots, D_n)$ be a document collection, $I_{D_{id}}$ be an index for document D with unique identifier $D_{id} = \mathsf{id}(D)$ and τ_w be a trapdoor for keyword w. Furthermore, the PRF F is defined as

$$\mathsf{F}\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$$

and the number of used PRFs is denoted by r.

Given this notation, the algorithms involved in the Z-IDX index are specified as follows:

`KeyGen`(1^k)**:** Given security parameter k:

1. Generate a master key $K = (K_1, \ldots, K_r) \xleftarrow{\$} (\{0,1\}^k)^r$, i.e., a sequence of r PRF keys

`BuildIndex`(K, D)**:** Given the master key $K = (K_1, \ldots, K_r)$ and document D including a unique identifier $D_{id} = \mathsf{id}(D)$ and a list of associated keywords $\hat{w} = (w_1, \ldots, w_q)$, initialise a Bloom filter $\mathcal{BF}$ for document D. Perform the following computations:

1. Build the index, i.e., for each unique keyword $w_i \in \hat{w}$:
   a) Compute trapdoor: $(x_1 \leftarrow \mathsf{F}(K_1, w_i), \ldots, x_r \leftarrow \mathsf{F}(K_r, w_i))$ where $|x_i| = k$ for $i \in [1,r]$
   b) Compute codeword: $(y_1 \leftarrow \mathsf{F}(x_1, D_{id},), \ldots, y_r \leftarrow \mathsf{F}(x_r, D_{id}))$ where $|y_i| = k$ for $i \in [1,r]$
   c) Insert the codeword $(y_1, \ldots, y_r)$ into the Bloom filter $\mathcal{BF}$ of document D_{id}
2. Blind the index:
   a) Estimate an upper bound u for the number of keywords for D. Since keywords could be one byte long, u is set to the number of bytes of the encrypted document D, i.e., the length of D's ciphertext.
   b) Given the number v of unique words in $\hat{w}$, set $(u - v) \cdot r$-times a bit to 1 at a uniformly random chosen position, each position could potentially be chosen multiple times.
3. Output the index $I_{D_{id}} = (D_{id}, \mathcal{BF})$ for document D with unique identifier D_{id}

`Trapdoor`(K, w)**:** Given the master key $K = (K_1, \ldots, K_r)$ and the keyword w, compute and output trapdoor $\tau_w \leftarrow (\mathsf{F}(K_1, w), \ldots, \mathsf{F}(K_r, w))$ for keyword w

`Search`$(I_{D_{id}}, \tau_w)$**:** Given index $I_{D_{id}} = (D_{id}, \mathcal{BF})$ for document $D_{id} = \mathsf{id}(D)$ and the trapdoor $\tau_w = (x_1 = \mathsf{F}(K_1, w), \ldots, x_r = \mathsf{F}(K_r, w)))$ for keyword w, document D_{id} can be searched for keyword w:

1. Compute codeword: $(y_1 \leftarrow \mathsf{F}(x_1, D_{id}), \ldots, y_r \leftarrow \mathsf{F}(x_r, D_{id}))$ where $|y_i| = k$ for $i \in [1,r]$
2. Test if the codeword $(y_1, \ldots, y_r)$ is stored in $\mathcal{BF}$, i.e., all r bits are set to 1
3. If all relevant bits are set: return 1, else return 0

BuildIndex In the `BuildIndex` algorithm, the forward index $I_{D_{id}}$ for a given single document with unique identifier $D_{id} = \mathsf{id}(D)$ and associated keywords $\hat{w} = (w_1, \ldots, w_q)$ is computed. Instead of inserting

a keyword directly into the Bloom filter, a trapdoor is computed using the PRF F with r different keys $K_1, \ldots, K_r$. Afterwards, the trapdoor is blinded utilising the PRF F and the unique document identifier. Finally, the blinded trapdoor known as codeword is inserted into the Bloom filter $\mathcal{BF}$. Obviously, this procedure has to be done for each unique keyword $w_i \in \hat{w}$. While the Bloom filter would be perfect usable after the insertion process, it is blinded by setting $(u - v) \cdot r$ bits at random positions to 1 where u denotes an upper bound for tokens in D and v represents the number of unique keywords in D. If a bit position should be randomly chosen more than once, nothing has to be done since it is already set, but no alternative position is sampled. Hence, this blinding routine produces the same outcome as inserting $u - v$ random words.

Trapdoor Examining the `Trapdoor` algorithm, it is easy to see that the trapdoor τ_w of a given keyword w is calculated the same way as in the `BuildIndex` algorithm, i.e., using the PRF F with r different keys and the keyword w as input.

Search The `Search` algorithm takes the trapdoor τ_w for keyword w and searches the index $I_{D_{id}}$ for occurrence of w in document D with unique identifier D_{id}. For this task, the codeword has to be computed analogous to the `BuildIndex` algorithm, i.e., using the PRF F with the trapdoor τ_w and the document identifier D_{id} as inputs. Afterwards, it is tested if the resulting codeword $(y_1, \ldots, y_r)$ is contained in the Bloom filter. If all r positions computed by the Bloom filter's hash functions are set, the keyword w is included in the index, otherwise not.

Secure Index Scheme

Based on Z-IDX, the secure index scheme can be constructed straightforward by computing the `BuildIndex` algorithm for all documents of the document collection. Beforehand, suitable parameters for the Bloom filter have to be chosen, i.e., the number of PRFs and the size of the underlying bit array. Both values depend on the desired false positive rate and the estimated number of unique words among all documents. The `KeyGen` algorithm has to be executed just once and the document collection has to be encrypted using a conventional encryption algorithm.

To search for a keyword, computing a single trapdoor using the `Trapdoor` is sufficient since the same master key and Bloom filter parameters have been used for all indices. Using this trapdoor, the server has to execute the `Search` algorithm for all stored documents.

4.1.3. Analysis

In the following, the most important aspects of Z-IDX and the secure index scheme are analysed.

Security

Examining the Z-IDX construction starting with focus on keywords, it can be seen that no dictionary or other similar concept for numbering and gathering keywords is used. Therefore, the universe of keywords does not have to be defined beforehand and it is possible to use all valid strings as keywords.

Considering the trapdoor and codeword generation, it can be seen that the keywords are not inserted in the Bloom filter directly, but the PRF F is used to calculate a codeword out of a trapdoor and a document

identifier. Hence, the indices for documents associated with the same keywords are different since the unique document identifier is included in the codeword computation. Therefore, this technique aims to prevent correlation attacks. Another benefit of this construction is the document independent trapdoor that can be used for the whole document collection. As usual, the advantage in terms of space requirement of transmitting such a short trapdoor comes at the price of additional computational effort since given the trapdoor, the server has to calculate an individual codeword for each document.

Even though the indices for same keywords are always different, the Bloom filter is blinded by adding random words. While the number of keywords included in the Bloom filter can only be estimated, adding additional unrelated words further obfuscates both the actual keywords and the number of keywords at costs of an increased false positive rate.

Regarding the overall security, Goh proved the scheme to achieve semantic security against an adaptive chosen-keyword attack (IND1-CKA). As already discussed before (see Section 3.4.1), the IND1-CKA definition has been shown to be insufficient. As stated later [KP13], the secure index scheme achieves the subsequently introduced non-adaptive indistinguishability for Searchable Symmetric Encryption (IND-CKA1). Consequently, this scheme is classified as secure in the non-adaptive setting.

Updates

As a consequence of using Bloom filters, updating existing indices is not always possible. If the update operation consists only of adding keywords, then it is possible to simply add these keywords as the number of set bits can never decrease. Since the keywords can not be recovered from the index and it is not known if a bit was set by a specific keyword, it is not possible to delete keywords efficiently. Hence, in case of an update that includes at least one deletion operation, the whole index has to be rebuild. Since all indices are independent, adding or deleting whole documents can be done in a straightforward manner.

Efficiency

Examining the efficiency, it can be seen that the index generation takes $O(q)$ time for each document where q specifies the number of distinct keywords per document, resulting in $O(q \cdot n)$ for a document collection of n documents. The trapdoor generation for a single keyword can be performed in $O(1)$ and a single keyword search needs $O(1)$ time per document resulting in $O(n)$ for searching n files. Due to the forward index construction, all indices can be processed individually allowing a high-grade parallelisation of the index generation and keyword search. Consequently, the indices can be build in $O(\frac{q \cdot n}{p})$ and a single keyword search can be performed in $O(\frac{n}{p})$ parallel search time where p indicates the number of processors (cores).

Space Requirements

As a result of using Bloom filters of fixed size of m bits in a simple construction, this scheme achieves low storage costs for indices in typical settings, i.e., $O(n)$ for $\boldsymbol{\mathcal{D}}$. Additionally, the index can be compressed by storing only the relevant bit positions instead of the whole Bloom filter array. Since no dictionary has to be stored, only a few kilobytes storage space per index for documents containing a few thousand keywords are needed.

Summary

While the secure index scheme is clearly not optimal in terms of search time, it provides a simple, space efficient construction that achieves reasonable security guarantees.

4.2. PPSED Scheme

In this section, the SSE scheme developed by Chang et al. is presented [CM05]. Following the notation of the defined problem setting as Privacy Preserving Keyword Searches on Remote Encrypted Data (PPSED), the scheme is called PPSED scheme in this thesis.

4.2.1. Idea

The PPSED scheme can be classified as a symmetric, forward index-based searchable encryption scheme. The general idea behind this scheme is using a dictionary listing all possible keywords for building a dictionary based index for each document. In order to avoid information leakage, this index is built using a pseudorandom permutation (PRP) and masked using PRFs resulting in different indices for identical documents.

Even though the scheme is based on a remarkable simple constructions, it achieves reasonable security and efficiency. As an outlook, the PPSED scheme requires search time linear in the document collection size. The space requirements depends on the dictionary size, but even if a full English dictionary is used, the resulting index needs only a few kilobytes storage space independent of the document size.

4.2.2. Construction

The construction of the PPSED scheme utilises a dictionary Δ to build the index. Extending the general definition of a dictionary from Section 3.3, in the context of the PPSED scheme, Δ defines d integer-keyword pairs (i, w_i) with $i \in [d], w_i \in \{0,1\}^*$ for some constant $d \in \mathbb{N}$.

Given a security parameter k, a document collection $\boldsymbol{\mathcal{D}} = (D_1, \ldots, D_n)$ and the dictionary Δ, the PRFs F, G and the PRP P are defined as following:

$$\begin{aligned}
\mathsf{F}&\colon \{0,1\}^k \times [d] \to \{0,1\}^k \\
\mathsf{G}&\colon \{0,1\}^k \times [n] \to \{0,1\} \\
\mathsf{P}&\colon \{0,1\}^k \times [d] \to [d]
\end{aligned}$$

In the following, the algorithms involved in the PPSED scheme are defined[1].

[1] Chang et al. have not divided their scheme into different algorithms. To provide a uniform representation and to keep things as simple as possible, the PPSED scheme is defined in accordance to the unified model for SSE schemes used in this thesis.

`KeyGen`(1^k)**:** Given security parameter k, generate the master key $K = (K_s, K_r)$ with $K_s, K_r \xleftarrow{\$} \{0,1\}^k$

`BuildIndex`$(K, \mathcal{D})$**:** Given the master key $K = (K_s, K_r)$ and the document collection $\mathcal{D} = (D_1, \ldots, D_n)$:

1. For all $i \in [d]$: Compute $K_{r_i} \leftarrow \mathsf{F}(K_r, i)$
2. For each $D_j \in \mathcal{D}$ with $1 \leq j \leq n$:
   a) Initialise I_j as array of d bits
   b) Build the index: For all $i \in [d]$: $I_j[\mathsf{P}(K_s, i)] = \begin{cases} 1 & \text{if document } D_j \text{ contains keyword } w_i \\ 0 & \text{else} \end{cases}$
   c) Mask the index: For all $i \in [d]$: $I_j[i] = I_j[i] \oplus \mathsf{G}(K_{r_i}, j)$
   d) Output the masked index I_j

`Trapdoor`(K, w_λ)**:** Given the master key $K = (K_s, K_r)$ and the keyword w_λ:

1. Retrieve the corresponding integer λ from Δ
2. Compute $p \leftarrow \mathsf{P}(K_s, \lambda)$ and $f \leftarrow \mathsf{F}(K_r, p)$
3. Output trapdoor $\tau_{w_\lambda} = (p, f)$

`Search`(I_j, τ_{w_λ})**:** Given index I_j for document D_j and the trapdoor $\tau_{w_\lambda} = (p, f)$ for keyword w_λ, document D_j can be searched for keyword w_λ:

1. Compute $X_j[p] = I_j[p] \oplus \mathsf{G}(f, j)$
2. If $X_j[p] = 1$ then w_λ is associated with D_j, return corresponding identifier $\mathsf{id}(D_j)$

BuildIndex In the `BuildIndex` algorithm, the forward indices $(I_1, \ldots, I_n)$ for document collection $\mathcal{D}$ consisting of n documents each associated with an arbitrary number of keywords from Δ are built. Each index has the size of d bits, correlating to the number of d entries in the dictionary. Each keyword in the dictionary is mapped to a unique pseudorandom position in the index by using a PRP that depends on the keyword position and the secret key K_s. If the respective keyword is part of the document, the bit is set in the corresponding index. After all keywords have been processed, the index is masked by XORing each index bit with a pseudorandom bit that depends on the document number, the keyword position and the secret key K_r.

Trapdoor The first task of the `Trapdoor` algorithm is to recover the keyword position from the dictionary. Using the keyword position and the master key, the pseudorandom position in the index and key used during the blinding process can be computed and output as trapdoor.

Search The trapdoor is used during the `Search` algorithm to check if the relevant index bit was set. For this task, the XOR operation done during the build process masking the index is reverted using the trapdoor to reveal the unmasked bit. As a note, Chang et al. defined the PPSED scheme for searching $\mathcal{D}$. Due to the adapted scheme definition matching the unified model used in this thesis, `Search` performs a single index search. If $\mathcal{D}$ shall be searched, `Search` has to be invoked n times.

4.2.3. Analysis

In this section, a brief analysis of the PPSED scheme is provided.

Security

Examining the keyword security, it can be seen that the number of keywords included in the index is hidden by the masking process, i.e., the XOR operation which depends on PRFs using the document number, the keyword position and the secret key. Since the document number is included in this process, the resulting indices of documents containing the same keywords are clearly not identical preventing correlation attacks. Even though the indices are blinded using the document identifier, the trapdoors are independent of the document identifier allowing short trapdoors that are usable to search all indices created under the same master key.

Regarding the overall security, the PPSED scheme was initially proven to achieve the semantic security against an adaptive chosen-keyword attack (IND2-CKA) model, but as already mentioned (see Section 3.4.1), the IND2-CKA definition has been shown to be not sufficient to obtain secure schemes. As indicated later [KP13], the PPSED scheme achieves the subsequently introduced IND-CKA1 security. Consequently, this scheme is classified as secure in the non-adaptive setting.

Updates

It is easy to see that the `BuildIndex`, `Trapdoor` and `Search` algorithm do not use any probabilistic methods. As consequence of the deterministic construction, keywords can be recovered if the master key is known. Therefore, assuming the dictionary stays unchanged, it is possible to perform efficient updates on arbitrary indices. If a keyword should be added or deleted from an index, the specific pseudorandom bit has to be updated accordingly.

Efficiency

Examining the efficiency, it can be seen that the index creation for a single document depends on d and the number of keywords in the document. Since the number of keywords in the dictionary is expected to be much larger than the number of keywords per document, i.e., $d \gg q$, `BuildIndex` needs $O(d \cdot n)$ for a document collection of n files. The trapdoor generation for a single keyword can be computed in $O(1)$ and a single keyword search needs $O(1)$ time per document resulting in $O(n)$ for searching n files. Due to the forward index construction, both the index generation and the keyword search can be parallelised. Consequently, building the indices can be done in $O(\frac{d \cdot n}{p})$ and a single keyword search needs $O(\frac{n}{p})$ time where p indicates the number of processors (cores).

Space Requirements

Considering the space requirements, a distinction between server and client has to be done. On the clientside, the dictionary is needed to perform searches. Obviously, the dictionary size depends on the number of entries and the average keyword length. On the serverside, the storage space for the indices depends on d, i.e., the number of keywords in the dictionary. Therefore, for a document collection of n documents, $O(d \cdot n)$ space is needed.

To provide an example, a full English dictionary would result in a storage need of about 2 megabytes, making the additional storage overhead compared to schemes that do not need a dictionary quite negligible. Using this dictionary, d would be 2^{18} resulting in about 32 kilobytes per index while there are still more that 37000 free positions for new keywords available. By applying compression techniques, this index size can be further reduced. If the dictionary storage space should not be available locally, Chang et al. proposed a slightly modified scheme that avoids this issue at the cost of increased communication complexity. The modified scheme stores the dictionary on the server, where the keyword of all integer-keyword pairs is encrypted. For computing a trapdoor, an additional communication round is needed to retrieve the integer λ corresponding to the encrypted keyword.

Summary

While the upcoming constructions achieve better security guarantees, the PPSED scheme provides a reasonable tradeoff between space efficiency and search performance.

4.3. SSE-1 Scheme

In [Cur+06] Curtmola et al. present two SSE constructions and the already introduced IND-CKA1 and IND-CKA2 security definitions that can be seen as a de facto standard (see Sections 3.4.3 and 3.4.4). While the first construction called SSE-1 achieves non adaptive IND-CKA1 security, the second one (SSE-2) achieves the stronger IND-CKA2 security notion at a price of increased storage and communication costs. In the following, the more efficient SSE-1 scheme is presented. Readers interested in the very similar SSE-2 solution are referred to [Cur+06].

4.3.1. Idea

The SSE-1 scheme can be classified as a symmetric, inverted index-based searchable encryption scheme. As implied by the inverted index type, the concept behind this scheme is to build an index per document collection where for each keyword a linked list of corresponding document identifiers is stored. The inverted index consists of an array storing the linked lists in a scrambled, obfuscated way and of a look-up table to find the first entry of each linked list. As initially stated, the authors proved this construction to be IND-CKA1 secure.

Regarding the efficiency, it can bee seen that due to the usage of linked lists for each keyword, only relevant entries have to be processed, resulting in a sublinear search time. In fact, the search time for a single keyword search is linear in the number of documents containing the keyword, i.e., $O(|\boldsymbol{\mathcal{D}}(w)|)$ which is clearly optimal. As one would might expect, the security and optimal search time comes at a price of increased storage costs in comparison to other schemes. As an outlook, the size of the inverted index is linear in the size of the document collection and the number of keywords that might occur, i.e., the number of entries in the dictionary.

4.3.2. Construction

Subsequently, for an array A, the function $\mathsf{addr}_A(x)$ gives the address of an element x in A, i.e., if $A[i] = x$, then $\mathsf{addr}_A(x) = i$. Furthermore, a linked list L stored into A consists of j nodes $N_i = \langle str_i, \mathsf{addr}_A(N_{i+1}) \rangle$

with $1 \leq i \leq j$ where str_i is an arbitrary string and $\mathsf{addr}_A(N_{i+1})$ denotes the address in A where the next node is stored. Considering the dictionary Δ, all keywords in Δ need at most ℓ bits storage space.

Additionally, let k be a security parameter, s be the size of the encrypted document collection measured in the smallest possible keyword size (for example one byte), A be an array with s entries and T be a $(\{0,1\}^{\ell} \times \{0,1\}^{k+\log_2 s} \times |\Delta|)$ look-up table. All entries in A are nodes of a linked list, all entries in T are of the form $\langle address, value \rangle$. The PRF F and the PRPs P, Q are defined as following:

$$\mathsf{F}\colon \{0,1\}^k \times \{0,1\}^{\ell} \to \{0,1\}^{k+\log_2 s}$$
$$\mathsf{P}\colon \{0,1\}^k \times \{0,1\}^{\ell} \to \{0,1\}^{\ell}$$
$$\mathsf{Q}\colon \{0,1\}^k \times \{0,1\}^{\log_2 s} \to \{0,1\}^{\log_2 s}$$

Given these definitions and a pseudorandomness against chosen-plaintext attack (PCPA) secure encryption scheme SKE, the algorithms involved in the SSE-1 scheme are defined as follows:

$\mathtt{KeyGen}(1^k)$**:** Generate and output the master key $K = (K_1, K_2, K_3)$ with $K_1, K_2, K_3 \xleftarrow{\$} \{0,1\}^k$

$\mathtt{BuildIndex}(K, \boldsymbol{\mathcal{D}})$**:** Given the master key $K = (K_1, K_2, K_3)$ and the document collection $\boldsymbol{\mathcal{D}} = (D_1, \ldots, D_n)$:

1. Initialisation: Compute $\Delta' = \delta(\boldsymbol{\mathcal{D}})$, compute $\boldsymbol{\mathcal{D}}(w)$ for all $w \in \Delta'$ and set global counter ctr to 1
2. Build array A: For $1 \leq i \leq |\Delta'|$:
   a) Generate key $K_{i,0} \leftarrow \mathtt{SKE.Gen}(1^k)$
   b) For $1 \leq j \leq |\boldsymbol{\mathcal{D}}(w_i)| - 1$:
      i. Let $\mathsf{id}(D_{i,j})$ be the j$^{\text{th}}$ document identifier in $\boldsymbol{\mathcal{D}}(w_i)$
      ii. Generate key $K_{i,j} \leftarrow \mathtt{SKE.Gen}(1^k)$
      iii. Create node $N_{i,j} = \langle \mathsf{id}(D_{i,j}) || K_{i,j}, \mathsf{Q}(K_1, ctr+1) \rangle$
      iv. Encrypt and store node $N_{i,j}$: $A[\mathsf{Q}(K_1, ctr)] \leftarrow \mathtt{SKE.Enc}_{K_{i,j-1}}(N_{i,j})$
      v. Increase counter: $ctr = ctr + 1$
   c) For the last node, i.e., $j = |\boldsymbol{\mathcal{D}}(w_i)|$:
      i. Create node $N_{i,j}$ with address of next node set to $NULL$: $N_{i,j} = \langle \mathsf{id}(D_{i,j}) || 0^k, NULL \rangle$
      ii. Encrypt and store node $N_{i,j}$: $A[\mathsf{Q}(K_1, ctr)] \leftarrow \mathtt{SKE.Enc}_{K_{i,j-1}}(N_{i,j})$
      iii. Increase counter: $ctr = ctr + 1$
3. Blind array A: If $q' < s$ with $q' = \sum_{w_i \in \Delta'} |\boldsymbol{\mathcal{D}}(w_i)|$, then set the remaining $s - q'$ entries of A to random values of same size as existing ones
4. Build look-up table T: For all $w_i \in \Delta'$: $T[\mathsf{P}(K_3, w_i)] = \langle \mathsf{addr}_A(N_{i,1}) || K_{i,0} \rangle \oplus \mathsf{F}(K_2, w_i)$
5. Blind look-up table T: If $\Delta' < \Delta$, then set the remaining $\Delta' - \Delta$ entries of T to random values of same size as existing ones
6. Output I with $I = (A, T)$

$\mathtt{Trapdoor}(K, w)$**:** Given the master key K and the keyword w: Output trapdoor $\tau_w = (\mathsf{P}(K_3, w), \mathsf{F}(K_2, w))$

`Search`(I, τ_w): Given index I and the trapdoor $\tau_w = (t_w, b_w)$ for keyword w:

1. If $T[t_w] = \perp$: Return $\perp$, else:
   a) Parse $T[t_w] \times b_w$ as $\langle adr || K' \rangle$
   b) Decrypt all nodes of the linked list starting with node stored at address adr with key K'
   c) Output the document identifiers of all processed nodes

BuildIndex At first $\mathcal{D}$ has to be scanned for unique keywords Δ' and for each found keyword, a list of document identifiers corresponding to the keyword's appearances has to be generated, i.e., $\mathcal{D}(w_i)$ for all $w_i \in \Delta'$. This information could be provided or computed in other ways, for example if multimedia content is included where simple scanning is not possible. Afterwards, for each keyword w_i, a linked list of $|D(w_i)|$ nodes is built and stored into the array A in an encrypted way at random positions calculated by PRP Q. To be precise, for keyword w_i, $|\mathcal{D}(w_i)|$ nodes are generated, where each node contains a document identifier, the key and address of the next node. Each node is encrypted with the key saved in the previous node and stored at a random address also saved in the previous node. In the last node of each list, the address of the next node is set to $NULL$ to mark the end of the linked list. Afterwards, if $q' < s$ with q' being the sum over all documents of the number of distinct keywords in each document and s being the total size of the encrypted document collection measured in minimum keywords size, A is blinded by inserting $s - q'$ entries with random values having the same size as the existing ones to A.

To find the first node of the linked list for keyword w_i stored in A, a look-up table T is needed. Therefore, for each distinct keyword $w_i \in \Delta'$, a look-up table entry storing the address of the first node of the corresponding linked list and the key needed to encrypt the node has to be created. All look-up table entries are obfuscated by XORing the concatenation of address and key with the output of PRF $\mathsf{F}(K_2, w_i)$ as blinding value. Additionally, all entries are stored at random positions calculated using PRP $\mathsf{P}(K_3, w_i)$. Similar to the blinding of A, if $|\Delta'| < |\Delta|$, T is blinded by setting $|\Delta| - |\Delta'|$ entries to random values of same size as the existing ones.

Trapdoor Regarding the search for a keyword, it is quite obvious that the corresponding look-up table position has to be recovered. Therefore, the `Trapdoor` algorithm taking keyword w as input needs to compute the position $\mathsf{P}(K_3, w)$ in T and the blinding value $\mathsf{F}(K_2, w)$.

Search The `Search` algorithm receiving trapdoor $\tau_w = (t_w, b_w)$ and index I searches the whole document collection for keyword w. If an entry at $T[t_w]$ is found, the first node address and corresponding key can be recovered by XORing b_w with the value at $T[t_w]$. Afterwards, starting with the found node, the document identifiers related to the searched keyword can be retrieved by iterating through the linked list and decrypting all nodes until the node with $NULL$ as next address occurs.

4.3.3. Analysis

In the following, the SSE-1 construction is analysed.

Security

Examining the array A of the index construction, it can be seen that each encrypted node is stored at a random position in A computed by PRP Q. As a result, all linked lists are stored in a scrambled form into A hiding the number of lists and the length of each list. Since the number of lists corresponds to the number of distinct keywords and the lengths correspond to the number of matching documents, this information is not leaked from A as long as no relations between the encrypted nodes can be found. To achieve that nodes can not be correlated, each node is encrypted under a unique key stored in the previous node of the respective linked list. As final leakage preventing measure, A is blinded by inserting $s - q'$ entries if the total number of entries in A should be less than the size of the encrypted document collection measured in minimal keyword length. Without this blinding procedure, an adversary would learn q' and a rough estimator on the sum of multiple keyword appearances in $\mathcal{D}$, but details would not be leaked, i.e., which keywords appear more than once, the exact count for each keyword and which documents are affected.

Regarding the second part of the index, the look-up table T, the position and encryption of all entries in T depend on the respective distinct keyword $w_i \in \Delta'$. Hence, the total number of entries is the only information leaked from T, i.e., $|\Delta'|$, the number of distinct keywords in $\mathcal{D}$. To avoid this leakage, T is blinded by inserting $|\Delta| - |\Delta'|$ random entries so that the number of entries in T is always $|\Delta|$. Since Δ covers all possible keywords in respect to the maximum keyword length ℓ, the adversary is not able to learn which and how many keywords actually appear in $\mathcal{D}$.

As indicated by the leakage analysis of $I = (A, T)$, the adversary is only able to learn the access and the search pattern fulfilling the current most common informal requirement that nothing else should be revealed. Therefore and as proven by Curtmola et al., the SSE-1 scheme achieves the IND-CKA1 security notion.

Updates

An examination of the `BuildIndex`, `Trapdoor` and `Search` algorithm reveals that no probabilistic methods are used implying that keywords can be recovered if the master key is known. However, updating the index can not be done in a simple way. Considering the simple case of adding documents without new keywords, new nodes have to be added to the existing linked lists and the last node of the affected lists have to be updated. Furthermore, suitable positions in A have to be found and since s and q' have changed, the blind has to be adjusted. If keywords should be added or deleted or if documents shall be removed, the updating process would be even more complex. Therefore, updates can be considered as an expensive operation. Accordingly, Curtmola et al. recommend building a new index in case of an update, for example by recovering all required information from the original one and reusing it.

Efficiency

Regarding the efficiency of the index generation, it can be seen that for each keyword in each file a node in A has to be created, i.e $O(q')$ for $\mathcal{D}$ respectively $O(q)$ per document where q denotes the number of keywords in the document. Furthermore, for all keywords in $\mathcal{D}$, a look-up table entry has to be created, i.e., $O(|\Delta'|)$. Since it can be expected that the sum of all keywords over all documents is much larger than the number of distinct keywords, i.e., $q' \gg |\Delta'|$, `BuildIndex` takes $O(q')$ or roughly $O(q \cdot n)$ time. Since each linked list can be computed individually, the index generations can be parallelised, i.e., roughly $O(\frac{q \cdot n}{p})$ time where p indicates the number of processors (cores).

For the search process, one look-up in T has do be done, i.e., $O(1)$ if an efficient table storage format is used. Afterwards, all nodes of the found linked list have to be decrypted, i.e., $O(|\mathcal{D}(w)|)$ operations. Consequently, `Search` runs in $O(|\mathcal{D}(w)|)$ achieving optimal search time proportional to the number of found documents. As usual, the trapdoor provided by `Trapdoor` can be calculated in $O(1)$ time.

Space Requirements

After all blindings, A contains s entries with s being the size of the encrypted document collection expressed in minimal keyword length and T contains $d = |\Delta|$ entries. Consequently, the index has an asymptotical space requirement of $O(d+s)$. In practice, depending on the minimum keyword size and number of documents, it is not unlikely that the index could be larger than the document collection. As Ogata et al. have shown [Oga+13], if the document collection contains very short keywords but a rather large amount of files, the ratio $|I|\backslash|\mathcal{D}|$ could be over 80 in such extreme constellations. As second experiment shown in [Oga+13], a set of nearly 1000 papers presented at information security conferences achieves a $|I|\backslash|\mathcal{D}|$ ratio of about 28. Therefore, it is reasonable to assume that the index will be a lot larger than the document collection.

In search for improvements, a more detailed analysis reveals that A has s entries each having a size of $\lceil \log_2 n \rceil + k + \lceil \log_2 s \rceil$ bits. The look-up table T has d entries of $\ell + k + \lceil \log_2 s \rceil$ bits each. Assuming that the Advanced Encryption Standard (AES) is used for encrypting the nodes, k is at least 128 bits. If an anonymous cipher[2] would be used to encrypt the nodes, there would not be the need to apply a different key for each node saving k bits per node, i.e., $s \cdot k$ bits in total. As final thought and according to [Oga+13], a lot of space could be saved in average settings if A would not be blinded. Without a blind, A would contain q' instead of s entries introducing the leakage of q' and the knowledge if duplicate keywords in $\mathcal{D}$ exist.

Summary

All in all the SSE-1 scheme achieves strong security and optimal search time providing an all-purpose solution if storage space is not a critical concern. If IND-CKA1 is not sufficient, the very similar SSE-2 construction achieving the stronger IND-CKA2 security at a price of increased storage and communication costs could be considered. The SSE-2 scheme defines a label for each existing keyword-document combination by adding a keyword-document specific number to each keyword. Instead of storing all labels into A, they are stored directly into D at pseudorandom positions. To search for a keyword, n trapdoors have to be generated since for each keyword-document combination a different label has been used.

4.4. KRB Scheme

In this section, the tree based scheme developed by Kamara et al. [KP13] is presented. Subsequently, we call it keyword red-black (KRB) scheme.

[2] How to Search on Encrypted Data: Searchable Symmetric Encryption (Part 5). URL: http://outsourcedbits.org/2014/08/21/how-to-search-on-encrypted-data-searchable-symmetric-encryption-part-5/ (Accessed 10. December 2016)

4.4.1. Idea

The idea of the KRB scheme is to build an inverted index in form of a modified red-black tree called KRB tree, which is a special type of a height-balanced binary search tree. In this tree, the leaf nodes store pointers to the files and all internal nodes indicate for all possible keywords if there is at least one path to a leaf node that represents a file containing the respective keyword. Since subtrees can be processed individually, the scheme supports parallel search. Additionally, the tree structure allows to perform updates efficiently.

As an outlook, the KRB scheme achieves sublinear search time slightly above the optimal search time, i.e., $O\left(\frac{|\boldsymbol{\mathcal{D}}(w)|}{p} \cdot \log n\right)$ where p indicates the number of processors (cores). Additionally, updates can be done in $O(\log n)$ time and the underlying index in tree form needs $O(d \cdot n)$ storage space. Considering the overall security, this scheme is proven to be IND-CKA2 secure meaning that the strong adaptive security model is accomplished.

4.4.2. Construction

Before describing the actual scheme, the KRB tree generation needed for constructing the index has to be discussed.

KRB Tree

A red-black tree is a binary tree where all nodes are coloured in a way that all leaf nodes are black, the children of red nodes are always black and all paths from a given node to all its leaves contain the same number of black nodes. These properties guarantee that inserting, deleting and searching for elements can always be done in $O(\log n)$.

Using this data structure, the $\texttt{BuildTree}(\boldsymbol{\mathcal{D}})$ algorithm constructing the unencrypted KRB tree utilising a document collection $\boldsymbol{\mathcal{D}}$ and a dictionary $\Delta = (w_1, \ldots, w_d)$ of d words is defined as following:

$\texttt{BuildTree}(\boldsymbol{\mathcal{D}})$**:** Given the document collection $\boldsymbol{\mathcal{D}}$:

1. Order the identifiers of all documents in $\boldsymbol{\mathcal{D}}$, i.e., $\mathsf{id}(D_i) < \mathsf{id}(D_{i+1})$ for $i \in [1, n-1]$
2. Build a red-black tree T on top of the ordered identifiers and store file identifiers and pointers to the documents at the leaves
3. For each node u, store a d-bit vector $data_u$
4. For each leaf node $\ell \in T$ representing document D_l:
   a) For all $w_i \in \Delta$: $data_l[i] = \begin{cases} 1 & \text{if } D_l \text{ contains keyword } w_i \\ 0 & \text{else} \end{cases}$
5. For all internal nodes u with left child v and right child z, the $data_u$ vector is computed recursively from lower nodes to the root node by applying the bitwise OR disjunction as follows:

$$data_u = data_v \mid data_z$$

As can be seen in `BuildTree`, the *data* vector of each node consists of d bits where each bit represents a keyword in Δ, i.e., $data[i]$ relates to w_i. Consequently, if the i^{th} data bit of an arbitrary internal node u is set, i.e., if $data_u[i] = 1$, then there is at least one path to a leaf node that contains a document including keyword w_i.

To search for a keyword w_i, starting with the root node, the bit at position i has to be checked and if it is set, the child nodes have to be processed in the same way. At some point, the set of all reached leaf nodes storing all relevant document identifiers is returned. This search process takes $O(|\boldsymbol{\mathcal{D}}(w)| \cdot \log n)$ time and since subtrees can be processed individually, it can be done $O\left(\frac{|\boldsymbol{\mathcal{D}}(w)|}{p} \cdot \log n\right)$ time if parallelised where p indicates the number of processors (cores).

KRB Scheme

Once again, let $\boldsymbol{\mathcal{D}}$ be a document collection, k be a security parameter and $\Delta = (w_1, \ldots, w_d)$ be the dictionary. The PRF F, the PRP P and the random oracle O are defined as following:

$$\mathsf{F}\colon \{0,1\}^k \times \{w_1, \ldots, w_d\} \to \{0,1\}^k$$
$$\mathsf{P}\colon \{0,1\}^k \times \{w_1, \ldots, w_d\} \to \{0,1\}^k$$
$$\mathsf{O}\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}$$

Furthermore, λ denotes a keyword hash table of d $(key, value)$ tuples where key is from $\{0,1\}^k$ and $value$ is the encryption of a boolean value.

Given these definitions and an indistinguishability under chosen-plaintext attack (IND-CPA) secure encryption scheme `SKE` taking an additional randomness as input for the key generation `Gen`, the algorithms involved in the KRB scheme are defined as following:

$\texttt{KeyGen}(1^k)$**:** Generate and output the master key $K = (K_1, K_2)$ with $K_1, K_2 \xleftarrow{\$} \{0,1\}^k$

$\texttt{BuildIndex}(K, \boldsymbol{\mathcal{D}})$**:** Given the master key $K = (K_1, K_2)$ and the document collection $\boldsymbol{\mathcal{D}} = (D_1, \ldots, D_n)$:

1. Generate unencrypted tree: $T \leftarrow \texttt{BuildTree}(\boldsymbol{\mathcal{D}})$
2. Generate keys per keyword: For $1 \leq i \leq d : SK_i \leftarrow \texttt{SKE.Gen}(1^k, \mathsf{F}(K_2, w_i))$
3. Build the encrypted KRB Tree: For all nodes $v \in T$ with identifier $\mathsf{id}(v)$:
   a) Instantiate and store two (k,d) keyword hash tables $\lambda_{0,v}, \lambda_{1,v}$ at node v
   b) For $1 \leq i \leq d$:
      i. $b \leftarrow \mathsf{O}(\mathsf{P}(K_1, w_i), \mathsf{id}(v))$
      ii. $\lambda_{bv}[\mathsf{P}(K_1, w_i)] \leftarrow \texttt{SKE.Enc}_{SK_i}(data_v[i])$
      iii. Store a random string at $\lambda_{|1-b|v}[\mathsf{P}(K_1, w_i)]$
      iv. Delete vector $data_v$
4. Output T as index I

$\texttt{Trapdoor}(K, w)$**:** Given the master key K and w, output trapdoor $\tau_w = (\mathsf{P}(K_1, w), \texttt{SKE.Gen}(1^k, \mathsf{F}(K_2, w)))$

`Search`(I, τ_w)**:** Given index $I = T$ and the trapdoor $\tau_w = (t_P, t_{SK})$ for keyword w: Let r be the root node of T. Call `SearchHelper`(r), the algorithm `SearchHelper`(Node u) is defined as follows:

1. $b \leftarrow \mathsf{H}(t_P, \mathsf{id}(u))$
2. $x \leftarrow \texttt{SKE.Dec}_{t_{SK}}(\lambda_{bu}[t_P])$
3. if $x = 0$: return, else continue
4. if u is a leaf node add document identifier stored at leaf u to set of found identifiers, else call `SearchHelper`(v), `SearchHelper`(z) with v, z being the children of u

Output the set of document identifiers that have been found by calling `SearchHelper`(r)

Additionally, the KRB scheme consists of update algorithms `BuildUpdate` executed clientside and `PrepareUpdate`, `ApplyUpdate` run serverside defined as following:

`PrepareUpdate`$(I, op, i, \mathcal{C})$**:** Given the index $I = T$, the type of operation $op = \{insert, delete\}$, the identifier i of the document that shall be added or deleted and the set of ciphertexts $\mathcal{C}$:

1. Perform the structural update on T in respect to the type of operation op and the document identified by i
2. Let $T(u)$ be the subtree of T that has to been accessed during the structural update of T, i.e., $T(u)$ consists of all nodes that were needed to perform the update
3. Output $\tau_{updateHelp} = (op, i, T(u))$

`BuildUpdate`$(K, \tau_{updateHelp}, D_i)$**:** Given the master key K, the update information $\tau_{updateHelp} = (op, i, T(u))$ and the document D_i in case of an insertion:

1. If $op = insert$ encrypt D_i as C_i
2. Process the structural update on $T(u)$ and let $T'(u)$ be the new subtree
3. For all nodes $v \in T'(u)$ that have new or modified ancestors compared to $T(u)$:
   a) Change node identifier from $\mathsf{id}(v)$ to $\mathsf{id}(v')$
   b) Instantiate and store new two (k, d) keyword hash tables $\lambda_{0,v}, \lambda_{1,v}$ at node v
   c) For $1 \leq i \leq d$:
      i. $b \leftarrow \mathsf{H}(\mathsf{P}(K_1, w_i), \mathsf{id}(v'))$
      ii. $\lambda_{bv}[\mathsf{P}(K_1, w_i)] \leftarrow \texttt{SKE.Enc}_{SK_i}(data_v[i])$ where $data_v[i]$ is the updated vector
      iii. Store a random string at $\lambda_{|1-b|v}[\mathsf{P}(K_1, w_i)]$
4. Output $\tau_{update} = (T'(u), C_i)$

`ApplyUpdate`(I, τ_{update})**:** Given the current index $I = T$ and the update token $\tau_{update} = (T'(u), C_i)$:

1. Copy the new information from $T'(u)$ to already structural updated tree T
2. Update the set of ciphertexts $\mathcal{C}$
3. Output T as index I

BuildIndex This algorithm builds the inverted index I in form of a KRB tree for document collection $\mathcal{D}$. At first, an unencrypted KRB tree T is built upon the ordered file identifiers using the previously discussed `BuildTree` algorithm. Afterwards, the information stored in the $data_v$ vector of each node $v \in T$ is protected as following: At first, two (k,d) keyword hash tables $\lambda_{0,v}, \lambda_{1,v}$ are instantiated. For each keyword, a bit $b \in \{0,1\}$ is computed by a random oracle taking the pseudorandom permutation of the keyword and the node identifier as inputs. This bit indicates whether $\lambda_{0,v}$ or $\lambda_{1,v}$ is used for the current keyword-node combination. Therefore, for each keyword $w_i \in \Delta$, λ_{bv} stores in encrypted form if the keyword w_i is contained in some reachable leaf node, i.e., the encryption of $data_v[i]$ is stored at a pseudorandom position in λ_{bv}. In the other hash table, i.e., $\lambda_{|1-b|v}$, a random string is stored at the same position. After this procedure has been performed for all keywords, the $data_v$ has been stored in encrypted form split upon the two hash tables and has to be deleted now.

Trapdoor Regarding the search for a keyword, the bit b indicating which hash table is valid for the specific keyword and the position in λ_{bv} has to be known. Therefore, the `Trapdoor` algorithm taking a keyword w calculates the position $t_P \leftarrow \mathsf{P}(K_1, w)$, which is also needed to recover b, and $t_{SK} \leftarrow \mathsf{F}(K_2, w)$ needed as decryption key for the value stored at $\lambda_{bv}[t_P]$.

Search The `Search` algorithm receiving index $I = T$ and trapdoor $\tau_w = (t_P, t_{SK})$ searches the whole document collection for keyword w. Starting with the root node of T, the bit b for the current node u is recovered by using the random oracle O, i.e., $b \leftarrow \mathsf{O}(t_P, \mathsf{id}(u))$ where t_P is the position information from the trapdoor. Afterwards, the entry at position t_P in λ_{bu} can be decrypted using t_{SK} as key, i.e., $x \leftarrow \mathtt{SKE.Dec}_{t_{SK}}(\lambda_{bu}[t_P])$. If x should be set, the keyword is contained in the current node u. As a reminder, x indicates if there is at least one path to a leaf node that contains a document including keyword w. If u is a leaf, its identifier is added the set of found ones. Otherwise the process is repeated for both child nodes of u. At some point, all relevant leaf nodes have been visited and the algorithm returns the found document identifiers.

BuildUpdate The `BuildUpdate` algorithm run clientside computes new keyword hash tables for all nodes of the subtree $T(u)$ generated by `PrepareUpdate` that have new or modified ancestors. For each relevant node v, the two hash tables $\lambda_{0,v}, \lambda_{1,v}$ are computed in the same way as during the index generation, i.e., the encryption of $data_v[i]$ with $1 \leq i \leq d$ is stored at a pseudorandom position in λ_{bv} where the bit b is the output of the random oracle.

BuildUpdate Problem The issue with the textbook `BuildUpdate` algorithm as defined by Kamara et al. is that for all nodes which have to be updated, the hash table computations are based on the updated data vector $data_v$. While $data_v$ is obviously available for leaf nodes representing new or updated files, it has to be remembered that the original data vector has been deleted during the index generation and can not be recovered efficiently. Hence, $data_v$ is not available for all inner nodes. For a better understanding, a distinction whether a document is added or deleted can be made:

Add Document: If a document is added (or modified), the leaf node's new (or updated) data vector can be processed as usual. Afterwards, all hash tables from all nodes following the path from the parent node to the root have to be updated. However, it is not possible to perform the update because it is not known which keywords have been set by the other child node.

Delete Document: If a document is deleted, again all hash tables lying on path to the root node have to be updated. Once again this is not possible since it is not known which keywords are still included in the other subtree.

In search for possible solutions, it is mandatory that the plaintext data vector has to be known. Consequently, the simplest solution would be storing the KRB tree built by `BuildTree` in the first step of the `BuildIndex` algorithm in encrypted form on the server and reuse it for updates.

4.4.3. Analysis

In the following, an analysis of the KRB scheme is provided.

Security

Investigating the KRB tree construction done by `BuildIndex`, it can be seen that all entries of the *data* vector built by `BuildTree` are encrypted and stored at a pseudorandom position in one of the two hash tables that has been chosen randomly. Additionally, a random string is stored in the unused hash table at the same pseudorandom position achieving indistinguishability between the two entries. The position within the table is calculated using PRP P receiving the keyword as input and the hash table is randomly chosen in dependence of both the keyword and the node identifier. Therefore, even if two unencrypted vectors $data_u, data_v$ of nodes u, v are identical, the hash tables $(\lambda_{0u}, \lambda_{1u})$ and $(\lambda_{0v}, \lambda_{1v})$ are completely different. Consequently, the number of keywords actually contained in $\boldsymbol{\mathcal{D}}$, the number of documents including a keyword and similarities between documents are hidden.

As indicated by the leakage analysis of I, the only information an adversary can learn are the access pattern and the search pattern. In the case of the KRB scheme, this property persists even if the queries are generated based on the previous ones. Therefore and as proven by the authors, the KRB scheme achieves the strong IND-CKA2 security, i.e., security in an adaptive setting.

Updates

While Kamara et al. provide an update algorithm, the need of additional knowledge in form of the initial KRB tree has been argued. Assuming that all data vectors are available by using the presented or some other possible problem solution, updates can be performed efficiently in $O(\log n)$ and could be parallelised as well.

Efficiency

Considering the efficiency of the index generation, it can be seen that building the unencrypted KRB tree using `BuildTree` takes $O(d \cdot n)$ time. Afterwards, for all nodes $u \in T$, the $data_u$ vector has to be encrypted resulting in q encryptions and q random storage operations per node where q indicates the number of keywords contained in the document (or combination of documents in case of an internal nodes). Since the tree contains n leaf nodes and a red-black tree is a special form of a binary search tree, I contains a total of $2n-1$ nodes. Consequently, the `BuildIndex` algorithm needs $O(q \cdot n)$ time. Since each node can be processed individually, the index generation can be performed in $O(\frac{q}{p} \cdot n)$ where p indicates the number of processors (cores).

For trapdoor generation, `Trapdoor` takes $O(1)$ time as usual. In the search process, all paths from the root node to relevant leaf nodes have to be processed by decrypting a single entry in each hash table. Since the maximum height of a red-black tree RB is $O(\log |RB|)$ with $|RB|$ being the total number of nodes in the tree [Cor+09], the KRB tree T consisting of $2n-1$ nodes has a height of $O(\log n)$. Therefore, `Search`

needs $O(|\boldsymbol{\mathcal{D}}(w)| \cdot \log n)$ time where $|\boldsymbol{\mathcal{D}}(w)|$ accounts for the number of documents containing keyword w, i.e., the number of paths that have to be searched. As shown by the authors, the search can be performed in $O\left(\frac{|\boldsymbol{\mathcal{D}}(w)|}{p} \cdot \log n\right)$ parallel search time.

Space Requirements

For each of the $2n-1$ nodes, two (k,d) hash tables have to be stored. Therefore, the index I has a space requirement of $O(d \cdot n)$. While this seems efficient in an asymptotic sense, it has to be considered that each hash table entry is equal to the block size of the used cipher, resulting in huge indices in practice.

Summary

All together the KRB scheme achieves the strong adaptive IND-CKA2 security and sublinear search time slightly above the optimal search time. Since this scheme also allows efficient updates, it is suitable for various settings, as long as the significant space requirements can be tolerated.

4.5. OXT Scheme

Among other things, the work in [Cas+13] presents several SSE constructions. In this section, the Oblivious Cross-Tags (OXT) scheme supporting boolean queries is presented. Readers interested in the simpler schemes, suitable for single keyword searches but less secure for boolean queries, are referred to the original work.

4.5.1. Idea

The intention behind the OXT scheme is to provide a solution for searches over multiple keywords in form of general boolean queries. For single keyword searches, a special data structure called tuple-set or T-Set is used. In the T-Set, for each keyword a list of data tuples consisting of document identifier and trapdoor information is stored. The concrete instantiation of a T-Set is realised as hash tables with a fixed number of buckets where for each data tuple, both the bucket and the position within the bucket are chosen randomly. For multiple keyword searches, an additional data set named X-Set is used to check whether the found documents for the first keyword also satisfy the remaining query.

Even if the search query is of a boolean type and not just a single keyword, the OXT scheme achieves sublinear search time. To be precise, it is proportional to the number of documents containing the least frequent keyword which appears to be optimal, i.e., $O(q \cdot |\boldsymbol{\mathcal{D}}(w')|)$ where q represents the number of keywords in the query and w' indicates the least frequent keyword. Additionally, the search process can be parallelised. Regarding the security guarantees, the authors have adapted the IND-CKA2 security definition for their construction meaning that OXT is proven to be secure in the adaptive setting.

4.5.2. Construction

The inverted index of the OXT scheme is based upon a conventional data set named X-Set and a special purpose data structure called T-Set. Before discussing the actual scheme, the T-Set data structure is discussed.

T-Set

The T-Set storing a list of data tuples for each keyword is constructed as a kind of modified hash table with B buckets each having a size of S. Therefore, a T-Set is an array of size $B \times S$ where all entries have two fields called *label* and *value*. The elements in *label* have a length of k bits, the ones in *value* are $|s|+1$ bits long where s is a string of variable length defined later. As an outlook, in the later presented OXT scheme *value* will contain the encrypted document identifiers and information needed to process boolean search queries.

In the following, let k be a security parameter, k' be the key space of PRF F, $\Delta' = (w_1, \ldots, w_{d'})$ be the set of distinct words in the document collection and T be an array such that for all $w \in \Delta'$, $T[w]$ contains a list $t = (s_1, \ldots, s_{|T[w]|})$ of $|T[w]|$ strings. The number of elements in the longest list is denoted by max_w, i.e., $max_w = |T[w]|$ where $|T[w]| \geq |T[w_i]|$ for all $w_i \in \Delta' \setminus \{w\}$.

Additionally, the PRFs F, G and the random oracle O are defined as

$$\begin{aligned}
\mathsf{F}&\colon \{0,1\}^{k'} \times [1, max_w] \to \{0,1\}^k \\
\mathsf{G}&\colon \{0,1\}^k \times \{w_1, \ldots, w_{d'}\} \to \{0,1\}^{k'} \\
\mathsf{O}&\colon \{0,1\}^k \to [1,B] \times \{0,1\}^k \times \{0,1\}^{|s_i|+1},
\end{aligned}$$

where s_i is the i$^{\text{th}}$ string in $T[w]$.

Given these definitions, a T-Set instantiation ($\texttt{TSetSetup}, \texttt{TSetGetTag}, \texttt{TSetRetrieve}$) is defined as follows:

$\texttt{TSetSetup}(T, 1^k)$**:** Given array T and security parameter k:

1. Initialise array $TSet$ of size $B \times S$ with fields $label, value$ for each entry
2. Initialize array $Free$ of size B where each entry is an integer set initialised to $\{1, \ldots, S\}$
3. Generate key $K_T \xleftarrow{\$} \{0,1\}^k$
4. For all $w \in \Delta'$:
   a) $stag \leftarrow \mathsf{G}(K_T, w)$
   b) $t = T[w]$
   c) For $i = 1, \ldots, |t|$
      i. Let s_i be the i$^{\text{th}}$ string in t
      ii. $(b, L, K) \leftarrow \mathsf{O}(\mathsf{F}(stag, i))$
      iii. If $Free[b]$ should be an empty set, restart $\texttt{TSetSetup}$, otherwise choose $j \xleftarrow{\$} Free[b]$ and remove j from $Free[b]$
      iv. Set bit $\beta = \begin{cases} 1 & \text{if } i < |t| \\ 0 & \text{if } i = t \end{cases}$
      v. Store entry: $TSet[b,j].label = L$ and $TSet[b,j].value = (\beta || s_i) \oplus K$
5. Output $(TSet, K_T)$

$\texttt{TSetGetTag}(K_T, w)$**:** Given the key K_T and w, calculate $stag \leftarrow \mathsf{G}(K_T, w)$ and output it as trapdoor τ_w

`TSetRetrieve`$(TSet, \tau_w)$: Given $TSet$ and $\tau_w = stag$:

1. Initialise t as empty list, β as 1 and counter i as 1
2. While $\beta = 1$:
   a) $(b, L, K) \leftarrow \mathsf{O}(\mathsf{F}(stag, i))$
   b) Search for index j such that $A[b, j].label = L$
   c) Let β be the first bit of $A[b, j].value$ and s the remaining $\mathsf{f}(k)$ bits of $A[b, j].value$
   d) Add s to the list t and increment i
3. Output list t

As can be seen in `TSetSetup`, at first for each keyword appearing in $\boldsymbol{\mathcal{D}}$ a *stag* (abbreviation for "small tag") is generated. For all $|\boldsymbol{\mathcal{D}}(w)|$ documents containing the current keyword w, a random oracle is utilised to calculate b indicating which bucket is used, L representing the *label* information and K defining the round key needed for obfuscating the document identifier. Since the bucket is derived randomly, it could happen that the specific bucket is already full, i.e., an overflow appears requiring a rerun of `TSetSetup` with fresh keys. If there is at least one free slot available, the current string s_i – which will be a document identifier and helper information in the OXT scheme – is concatenated to a bit β – indicating if s_i is the last element of $\boldsymbol{\mathcal{D}}(w)$ – and XORed with the round key to calculate the *value* field. Afterwards, these values are stored in bucket b at a randomly chosen free slot. Consequently, the lists of strings are randomly distributed over all buckets at random positions within the buckets.

Considering the `TSetGetTag` algorithm, the *stag* information acting as a trapdoor required to find entries for a keyword is calculated. In the `TSetRetrieve` algorithm, $TSet$-entries are searched until an element with $\beta = 0$ is found indicating that the end of the hidden list of strings has been reached. For this task, the trapdoor $\tau_w = stag$ is utilised to recover the bucket b, the label L within the bucket and the round key K. Using this values, the stored information can be recovered by searching for the corresponding entry with label L in bucket b and XORing the found *value* with K. This procedure is repeated until the end of the list is reached.

At this point, the probability of bucket overflows and other details are worth contemplating. As shown in [Cas+13], the probability of a bucket overflow in any of the B bucket is at most $B \cdot (e/k)^S$ which equals $\frac{q'x}{S} \cdot (e^{1-1/x}/x)^S$ with $q' = \sum_{w_i \in \Delta'} |\boldsymbol{\mathcal{D}}(w_i)|$ and x being the space overhead. By adjusting the choices of B and S, the probability that `TSetSetup` has to be restarted can be reduced to be negligible even for small space overheads. Furthermore, it has to be noted that it is possible that a bucket contains two entries with the same label L. Therefore, it could be the case that the wrong element is processed resulting in wrong search results, but the probability of such an event can be estimated as $BS^2 2^{-k}$ which is negligible for practical instantiations since the key size is expected to be at least 128 bits.

To closure the discussion of the T-Set generation, it has to be mentioned that the T-Set could be used as replacement for other inverted index constructions, for example in the SSE-1 scheme. In the following, the OXT scheme based upon the T-Set construction is presented.

OXT Scheme

Once again, let $\boldsymbol{\mathcal{D}}$ be a document collection, k be a security parameter, Δ' be the keywords contained in $\boldsymbol{\mathcal{D}}$ and SKE be an IND-CPA secure encryption scheme. The OXT scheme uses a T-Set instantiation $(\texttt{TSetSetup}, \texttt{TSetGetTag}, \texttt{TSetRetrieve})$ as described before and two PRFs F, G defined as

$$\mathsf{F}\colon \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$$
$$\mathsf{G}\colon \{0,1\}^k \times \mathbb{Z}_p^* \to \mathbb{Z}_p^*,$$

where prime p is the order of a multiplicative written group $\mathcal{G}$ generated by g. Given these definitions, the KeyGen and BuildIndex algorithms are defined as follows:

KeyGen(1^k)**:** Generate and output the master key $K = (K_S, K_X, K_I, K_Z)$ with $K_S, K_X, K_I, K_Z \xleftarrow{\$} \{0,1\}^k$

BuildIndex$(K, \boldsymbol{\mathcal{D}})$**:** Given the master key $K = (K_S, K_X, K_I, K_Z)$ and the document collection $\boldsymbol{\mathcal{D}} = (D_1, \ldots, D_n)$:

1. Initialise T as empty array of $d' = |\Delta'|$ elements
2. Initialise $XSet$ as empty set
3. For all $w \in \Delta'$:
   a) Initialise t to an empty list and counter c to 0
   b) Calculate $K_E \leftarrow \mathsf{F}(K_S, w)$
   c) For all $ind \in \boldsymbol{\mathcal{D}}(w)$ in random order
      i. Calculate $xind \leftarrow \mathsf{G}(K_I, ind)$, $z \leftarrow \mathsf{G}(K_Z, w||c)$, $y = xind \cdot z^{-1} (\text{mod } p)$
      ii. Calculate $xtag = g^{\mathsf{G}(K_X, w) \cdot xind}$ and append $xtag$ to $XSet$
      iii. Calculate $e \leftarrow \texttt{SKE.Enc}_{K_E}(ind)$ and append (e, y) to t
      iv. Increment counter c
   d) Set $T[w] = t$
4. Build $(TSet, K_T) \leftarrow \texttt{TSetSetup}(T, k)$
5. Update master key K by appending K_T
6. Output $I = (TSet, XSet)$

While KeyGen and BuildIndex follow the standard definition of a SSE scheme, the Search algorithm requires an interaction between client and server. Additionally, the trapdoor generation is also done in Search. Obviously it would be possible to provide a separate Trapdoor algorithm, but due to the interactions between client and server, listing all tasks combined in one algorithm seems to be easier to present and understand.

$\texttt{Search}(K, \hat{w}, I)$: The client is given the master key $K = (K_S, K_X, K_I, K_Z, K_T)$ and a query $\hat{w} = (w_1, \ldots, w_q)$, the server is given the index $I = (TSet, XSet)$

1. Client: Calculate $stag \leftarrow \texttt{TSetGetTag}(\texttt{K}_\texttt{T}, \texttt{w}_\texttt{1})$ and send $stag$ to the server
2. Server: Receive $stag$, calculate $t \leftarrow \texttt{TSetRetrieve}(\texttt{TSet}, \texttt{stag})$ and send $|t|$ to the client
3. Client: Receive $|t|$ and initialise $xtoken$ as empty array of $|t|$ elements
4. Client: For $1 \leq c \leq |t|$:
   a) For $2 \leq i \leq q$: Calculate $xtoken[c,i] = g^{\mathsf{G}(K_Z, w_1 || c) \cdot \mathsf{G}(K_X, w_i)}$
   b) Set $xtoken[c] = xtoken[c,2], ..., xtoken[c,q]$
5. Client: Send trapdoor $\tau_{\hat{w}} = xtoken$ to the Server
6. Server: Receive $\tau_{\hat{w}}$ and for $1 \leq c \leq |t|$:
   a) Retrieve (e,y) from the c^{th} entry in t
   b) If for all $2 \leq i \leq q$: $xtoken[c,i]^y \in XSet$ then send e to the client
7. Client: Calculate $K_e \leftarrow \mathsf{F}(K_S, w_1)$ and for all received e: Output identifier $ind \leftarrow \texttt{SKE.Dec}_{K_e}(e)$

BuildIndex Considering `BuildIndex`, it can be seen that the constructed index consists of two sets called $TSet$ and $XSet$. Intuitively, the $TSet$ serves as inverted index for single keyword searches and the $XSet$ is used for processing queries of arbitrary length. To compute the array T as input for `TSetSetup` building $TSet$, an iteration over all keywords in $\mathcal{D}$ and all documents containing the current keyword w is required. Within this process, at first the current identifier ind is obfuscated as $xind$ using F_p. Next, a blind $z \in \mathbb{Z}_p^*$ is derived based on the keyword and on a counter to ensure that blind values are always different. Then, $xind$ is blinded as y by applying z^{-1} and $xtag$ (abbreviation for "cross tag") representing an obfuscated keyword-identifier combination is calculated using a group operation and appended to $XSet$. Finally, the current identifier ind is encrypted as e under a keyword specific key and the combination (e,y) consisting of encrypted identifier and precomputed inverted blind value is appended to $T[w]$. After all iterations are completed, `TSetSetup` is used to build the $TSet$ and the index consisting of $(TSet, XSet)$ is returned.

Search In the `Search` algorithm including the trapdoor generation, the first keyword w_1 is searched in the $TSet$ and all found documents are tested on containing the remaining $q-1$ keywords. Therefore, the client generates the $stag$ for the first keyword and the server responds with the amount of found identifiers $|t|$ by using `TSetRetrieve`. As a reminder, $|t|$ corresponds to the number of documents containing w_1, i.e., $|\mathcal{D}(w_1)|$. Consequently, the client calculates for each found identifier $q-1$ trapdoors for all remaining $q-1$ keywords and stores them into $xtoken[c]$ with $c \in [1, |t|]$ being the document counter. Each individual trapdoor is calculated using the obfuscated first keyword together with the document specific counter, the obfuscated identifier $xind$ and the keyword specific blind. After receiving the $xtoken$ array, the server has to check for all previously found documents matching w_1 if the remaining keywords are also included. Therefore, for each of these documents, the precomputed inverse blind y has to be retrieved and applied to document related trapdoors contained in $xtoken$. If all $q-1$ resulting values are included in $XSet$, the document contains all q keywords and the found decrypted identifier is returned to the client for decryption. Since it might not be clear why the search process is correct at first glance, we present an example.

Example 4.1 (OXT Search)**.** Let $y_c = xind \cdot z_c^{-1}$ be the inverted blind value for the c^{th} document containing keyword w_i. The corresponding $xtag$ is computed as $xtag_{w_i} = g^{\mathsf{G}(K_X, w_i) \cdot xind}$ and appended to $XSet$. During the search process, token $xtoken[c,i] = g^{z_c \cdot \mathsf{G}(K_X, w_i)}$ for keyword w_i is calculated. The following equation

$$\begin{aligned} xtoken[c,i]^{y_c} &= g^{z_c \cdot \mathsf{G}(K_X, w_i) \cdot y_c} \\ &= g^{z_c \cdot \mathsf{G}(K_X, w_i) \cdot xind \cdot z_c^{-1}} \\ &= g^{\mathsf{G}(K_X, w_i) \cdot xind} \\ &= xtag_{w_i} \end{aligned}$$

shows that $xtag_{w_i}$ is recovered correctly by the server and can be tested for occurrence in $XSet$.

4.5.3. Analysis

In this section, the OXT scheme is analysed.

Boolean Queries

The OXT scheme is able to process arbitrary boolean search queries. The construction provided so far tests if all keywords of a query are contained, i.e., a simple conjunctive query. This method can be extended for non-existence of keywords by testing if a keyword is not in $XSet$. To be precise, the server receives a boolean formula and inserts true or false depending whether a keyword is contained in $XSet$ or not. By applying this technique, arbitrary boolean queries can be processed as long as there is at least one non-negated term needed for `TSetRetrieve`. For the case of only negated terms, a special field containing all stored documents can be introduced.

Security

Starting with the T-Set construction done by `TSetSetup`, it can be seen that all input strings are stored in random buckets at random positions in obfuscated form. The bucket and the position within the bucket are calculated using a random oracle receiving the obfuscated *stag* – which is itself a obfuscation of the keyword – as input. Consequently, the number of keywords actually contained in $\mathcal{D}$, the number of documents containing a keyword and similarities between documents are hidden. The only information leaked from the T-Set alone is the total number of entries, which is equal to $q' = \sum_{w_i \in \Delta'} |\mathcal{D}(w_i)|$ being the total number of keywords in $\mathcal{D}$.

Considering the OXT scheme, it can be seen that the servers possibilities to reuse and combine trapdoors is limited. Since the trapdoors for the $XSet$ are constructed in dependence of the first keyword, calculating the intersection of trapdoors built for the $XSet$ is impossible. As an example, given queries (w_1, w_2) and (w_1', w_2'), calculating the intersection of documents matching $(w2, w2')$ is prevented. Regarding the intersection of first keywords, it is possible to compute the intersection of identifiers if two queries have different first keywords but same remaining keywords. Sticking to the previous example, the intersection of documents matching (w_1, w_1') can be computed if $w_2 = w_2'$. In any case, the knowledge whether two queries have the same first keyword and the number of documents matching the first keyword is always leaked. As final remark, if a query consists of q keywords, the intersection of documents containing (w_1, w_i) for all $i \in [2,q]$ can be calculated if the order of trapdoors is not permuted for all documents.

As indicated before, the only information an adversary can learn are the access pattern, the search pattern and the conditional intersection pattern emerging from multiple keyword queries. Consequently and as proven by the authors, the OXT scheme achieves the strong IND-CKA2 security adapted for the setting of multiple keyword queries.

Updates

Since the list of identifiers matching a specific keyword is split randomly across all buckets and positions within the buckets, updating the index can not be done without iterating over the lists. Considering the case of adding new documents, new elements have to be added to the lists and the last element of the affected lists have to be updated since the bit – indicating that the end of the list is reached – has to be changed. In case of deleting documents, the elements in the respective keyword list have to be rebuilt after the found identifier, since the counter has changed for the remaining entries. Consequently, the update can be run in $O(|\hat{w}| \cdot |\boldsymbol{\mathcal{D}}(\hat{w})|)$ time where $\hat{w}$ indicates the keywords in the document affected by the update.

Efficiency

In the `BuildIndex` algorithm, for each keyword-identifier combination the entries for $TSet$ and $XSet$ have to be computed. Afterwards, the $TSet$ is calculated by `TSetSetup` which creates an entry in $TSet$ for each input string. Therefore, the index generation can be done in $O(q')$ time where q' denotes the sum over all documents of the number of distinct keywords in each document, i.e., $q' = \sum_{w_i \in \Delta'} |\boldsymbol{\mathcal{D}}(w_i)|$. Since both the `BuildIndex` algorithm and the involved `TSetSetup` algorithm can compute entries individually, the index generation can be computed in $O(\frac{q'}{p})$ parallel time where p indicates the number of processors (cores).

If the search query consists of more than one keyword, it has to be stressed that the expected least frequent keyword has to be used as first keyword to reduce the number of found identifiers. Consequently, the number of trapdoor generations and equality tests can be minimised by choosing the first keyword wisely. Therefore, the authors recommend to store some information to differentiate whether a keyword is more or less frequent, for example by using a Bloom filter. Considering the actual search process for a query of q keywords, `TSetRetrieve` has to process $|\boldsymbol{\mathcal{D}}(w_1)|$ elements and afterwards $q-1$ trapdoors and equality tests have to be done for all found identifiers. Consequently, `Search` needs $O(q \cdot |\boldsymbol{\mathcal{D}}(w')|)$ time for a query of q keywords where w' indicates the least frequent keyword. Since both the trapdoor generations and the equality tests can be parallelised, `Search` needs $O(\frac{q \cdot |\boldsymbol{\mathcal{D}}(w')|}{p})$ parallel search time where p indicates the number of processors (cores) available on client and server. As final thought, for both `BuildIndex` and `Search` some expensive operations are involved during processing the $XSet$, limiting the performance in practice.

Space Requirements

Examining the space requirements, it can be seen that the index consists of $TSet$ and $XSet$. As already mentioned, $TSet$ contains $q' = \sum_{w_i \in \Delta'} |\boldsymbol{\mathcal{D}}(w_i)|$ elements. In $XSet$, q' entries are stored where the entry size depends on the choice of the group. Therefore, the index I has an asymptotic space requirement of $O(q')$.

Summary

All in all the OXT scheme achieves the strong adaptive IND-CKA2 security and optimal search time for single keyword queries. Furthermore, this scheme supports arbitrary boolean queries in search time proportional to the number of documents related to the least frequent keyword which appears to be asymptotically optimal as well. Since updates can be performed at moderate costs and the space requirements seem to be reasonable, the overall performance provided by the OXT scheme makes it a suitable choice for a wide range of practical applications.

Chapter 5

Implementation and Testing Environment

In the practical part of this thesis, the previously presented Searchable Symmetric Encryption (SSE) schemes have been implemented to evaluate their usability in practice. This chapter describes the implementation and test setting used to evaluate the performance.

5.1. Implementation

In order to realise a platform independent implementation that could be integrated into existing frameworks for building secure cloud storages like Archistar[1], Java™ has been chosen as programming language. As implementation of the required cryptographic functionality, the IAIK Provider for the Java Cryptography Extension (IAIK-JCE)[2] including the IAIK ECCelerate™[3] add-on is used as cryptographic provider. To keep the implementation free from dependencies, no other external Java™ libraries are used.

5.1.1. Secure Index Scheme

To determine the Bloom filter parameters r (number of hash functions) and m (size of the Bloom filter), a desired false positive rate can be specified (set to $0,01$ in the test setting). Even though additional blinding values were applied to the Bloom filter exactly as in the scheme's description, no false positives occurred during the tests. As pseudorandom function (PRF) involved in the Bloom filter construction, F is implemented as HMAC-SHA256 (keyed-hash message authentication code (HMAC) using Secure Hash Algorithm (SHA) as cryptographic hash function with a hash size of 256 bits). The Bloom filter is based upon an available implementation[4] where unused parameters have been removed.

[1] ARCHISTAR – A framework for secure distributed storage. GNU General Public License. URL: `http://archistar.at` (Accessed 10. December 2016)

[2] IAIK-JCE. URL: `http://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/JCA_JCE` (Accessed 10. December 2016)

[3] IAIK ECCelerate™. URL: `http://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/ECCelerate` (Accessed 10. December 2016)

[4] Java™ implementation of a Bloom filter by Magnus Skjegstad. URL: `https://github.com/MagnusS/Java-BloomFilter` (Accessed 10. December 2016)

5.1.2. PPSED Scheme

The PRFs F and G involved in the masking computation are implemented as HMAC-SHA256, where the least significant output byte of G is XORed with `0x01`. The pseudorandom permutation (PRP) P, computing the position of a keyword within the index, provides a mapping from an integer to another integer of the same range, i.e., from $[1,d]$ to $[1,d]$. For implementing P, typical block cipher modes are not suitable since the output would be a multiple of the block size and a modulo reduction is not possible because it would invalidate the bijection resulting in a PRF instead of a PRP. In practice, it would be possible to extend d to the block size (128 bits in case of the Advanced Encryption Standard (AES)) without reduction, but this would result in significantly larger index sizes because $d \ll 2^{128}$.

The first implemented solution is using a Knuth shuffle [Knu97], an algorithm taking a set of integers as input and returning them in shuffled order. To ensure the order is reproducible but not predictable, a secure random number generator initialised with key K_s is used. While this solution produces a valid PRP, the permutation has to be recalculated for each `Trapdoor` invocation or stored and safeguarded locally.

The second implemented approach adapts an available Java™ implementation[5] of AES in FFX[6] mode [BRS10], a mode for Format-Preserving Encryption (FPE) to achieve identical plaintext and ciphertext formants, i.e., the encryption of a n-bit number results in another n-bit number. While AES-FFX does not allow mapping integers of exactly same range, it reduces the space overhead since the number of digits stays the same. As a result, a dictionary containing d words results in an index of $10^{\lceil \log_{10} d \rceil}$ bits instead of d bits compared to the paper version of the Privacy Preserving Keyword Searches on Remote Encrypted Data (PPSED) scheme. To optimise the index computation, the masking is calculated for all used pseudorandom positions and the masking of all unused positions (introduced by the index length expansion) is set randomly. Therefore, all additional index bits are set randomly avoiding unexpected information leakage.

In the test setting, AES-FFX is selected as PRP P to minimise the difference between the scheme's original definition and implementation.

5.1.3. SSE-1 Scheme

In the SSE-1 scheme, the PRF F computes a blinding value that is XORed with a look-up table entry to obfuscate it. Since the output of F has to be exactly the same size as the look-up table entry, using HMAC-SHA256 in a straightforward way as previously is not possible because the entry's size would be limited resulting in a low upper limit of the supported size of $\mathcal{D}$. Instead, a Password-Based Key Derivation Function 2 (PBKDF2) [Nat10], which applies a PRF several times to calculate a key of requested length, could be used. Therefore, F is implemented as PBKDF2 applying HMAC-SHA256 as PRF.

In the paper version of the scheme, the position of the look-up table entry is calculated by PRP P where the output size depends on the maximum keyword length in $\mathcal{D}$. By using AES in Cipher Feedback (CFB) mode in the implementation, the output is a multiple of 128 bits resulting in a slight space increase of the look-up table compared to the original version.

Finally, PRP Q computing the next node position in the array defines a mapping from an integer to an integer of the same range. Following the same reasoning as in the PPSED scheme, Q is implemented as AES-FFX avoiding a space increase of the index since a Java™ `int` should be used anyway. For encrypting the nodes, AES-CFB is used as symmetric encryption scheme `SKE`.

[5] Java™ implementation of AES-FFX by Michael Tandy. URL: `https://github.com/michaeltandy/java-ffx-format-preserving-encryption` (Accessed 10. December 2016)

[6] The name FFX indicates Format-preserving, Feistel-based encryption with multiple parameter parameter choices [BRS10].

5.1.4. KRB Scheme

The keyword red-black (KRB) tree serving as index is based upon a red-black tree implementation from Sedgewick and Wayne[7]. In contrary to the paper version of the scheme, the plaintext data vectors are never part of the tree but only stored temporary in a separate data structure. Therefore, the KRB tree never contains any plaintext keyword information superseding the need to delete the plaintext data vectors preventing accidental information leakage.

The PRF F producing the input for the round key generations has a fixed output size of k bits, matching the key length of the symmetric encryption scheme `SKE` that is used to encrypt the plaintext vectors. As before, `SKE` is realised as AES-CFB. Consequently, PRF F is implemented as HMAC-SHA256, since k can not exceed 256.

The PRP P calculating positions within the hash table has the same key and output size, ideally suiting the straightforward usage of AES-CFB.

As final mentionable design decision, the random oracle O determining the active hash table for the current keyword-node combination is implemented as SHA256 where the least significant output byte is XORed with `0x01`.

5.1.5. OXT Scheme

For a better readability, the implementations of the T-Set and the actual Oblivious Cross-Tags (OXT) scheme are discussed separately.

T-Set

The T-Set parameters B (number of buckets) and S (size of each bucket) are calculated by a trial-and-error approach using a fixed required overflow probability (set to $0,001$ in the test setting) and space overhead (set to $1,1$ in the test setting). In the implemented algorithm, S is fixed and B is calculated accordingly. If the parameters lead to an estimated overflow probability not satisfying the desired one, S is increased in dependence of the current value (the increase scales with S) and the procedure is repeated. If an overflow should occur during the T-Set setup, the number of contained keywords is increased by ten percent for the retry to increase the probability of success.

As PRFs involved in the T-Set construction, the PRF G calculating the *stag* used as trapdoor and the PRF F using the *stag* and current position for calculating the random oracle input are both implemented as HMAC-SHA256. Per definition, the random oracle O outputs three values, namely b (the bucket), L (the label) and K (the round key used for blinding the entry). The random oracle is implemented as SHA256, where for each value a different fixed input prefix is used and the output is converted as intended. Since the round key K has to be of variable size, i.e., exactly one bit longer than the length of the current entry's value, a `SecureRandom` seeded with the oracle output is used to produce a variable length round key.

[7] Robert Sedgewick and Kevin Wayne: Algorithms, 4th Edition, Section 3.3 Balanced Search Trees. URL: `http://algs4.cs.princeton.edu/33balanced/` (Accessed 10. December 2016)

OXT Scheme

In the OXT scheme, the PRF F calculating the round key is implemented as PBKDF2 applying HMAC-SHA256 to support arbitrary key sizes. For the choice of the group, the prototype described in the original work [Cas+13] uses elliptic curves. Following the authors choice and fitting to the security level provided by HMAC-SHA256 and AES-CFB (with 128 bits key length), the National Institute of Standards and Technology (NIST) P-224 elliptic curve being a recommended curve over $\mathbb{F}_p$ with a prime size of 224 bits is used [Nat13]. For PRP P encrypting the current keyword, AES-CFB is used where the output is reduced modulo the curve order. Once again, the symmetric encryption scheme `SKE` encrypting the document identifiers is implemented as AES-CFB.

5.2. Parallel versus Sequential Execution

In our implementation, all algorithms are executed sequentially. However, the index generation of all schemes and the keyword search of both forward index-based SSE schemes (secure index and PPSED scheme), the KRB scheme and the OXT scheme could be parallelised. Consequently and apart from the `Trapdoor` algorithms, the `Search` algorithm of the SSE-1 scheme is the only one that has to be executed sequentially. Therefore, we believe that a sequential execution of all tests provides comparable results, but it has to be stressed that the search performance of the SSE-1 scheme would not profit from a high-grade parallelisation.

5.3. Test Setting

In this section, we present the setup for our test environment.

5.3.1. Document Collections

For testing the implemented schemes, the Reuters-21578 collection[8] being free to use for research purposes[9]has been selected as input for defining self-created document collections.

According to the documentation[9], the Reuters-21578 collection - in the following called Reuters collection – contains information that appeared on the Reuters newswire in 1987. Over the years, the collection's quality has been steadily improved with the objective of providing a standard test collection for text categorisation and machine learning classification tasks.

In the current version, a piece of news is specified by several attributes, for example date, title and message text. While the title can be taken as document identifier and the message text as document content, the associated keywords have to be extracted from the available information.

For this keyword extraction, the Moby Part-of-Speech word lists[10] have been used. Among other things, the Moby project contains several word lists ordered by word classes such as nouns, adverbs and adjectives.

[8]The Reuters-21578, Distribution 1.0 test collection is available from `http://www.daviddlewis.com/resources/testcollections/reuters21578`. (Accessed 10. December 2016)

[9]Readme of the Reuters-21578 including the copyright notice. URL: `http://www.daviddlewis.com/resources/testcollections/reuters21578/readme.txt` (Accessed 10. December 2016)

[10]The Moby project created by Grady Ward. URL: `http://icon.shef.ac.uk/Moby/` (Accessed 10. December 2016)

In order to generate simple keywords needed for constructing meaningful test queries, only nouns are allowed as keywords. Consequently, the message texts of all documents in the Reuters collection are scanned for nouns which are used as keywords for the respective document. Before the scanning task, all words in the document content and all words in the Moby noun list are converted to lowercase. This lowercase conversion ensures a consistent keyword formatting by reducing unintended keyword diversity due to formatting issues in the Reuters collection.

For a better understanding of the difficulties of keyword extraction, the following example showing raw input data and the generated keywords is presented:

- Title

```
ICO PRODUCERS TO PRESENT NEW COFFEE PROPOSAL
```

- Message Text

```
International Coffee Organization, ICO,producing countries will present a proposal
    for reintroducingexport quotas for 12 months from April 1 with a firmundertaking
    to try to negotiate up to September 30 any futurequota distribution on a new
    basis, ICO delegates said. Distribution from April 1 would be on an unchanged
    basis asin an earlier producer proposal, which includes shortfallredistributions
    totalling 1.22 mln bags, they said. Resumption of an ICO contact group meeting
    with consumers,scheduled for this evening, has been postponed until tomorrow,
    delegates said. Reuter&#3;
```

- Extracted Keywords

```
coffee, organization, countries, proposal, quotas, months, april, september,
    distribution, basis, producer, resumption, meeting, evening
```

It is easy to see that the Reuters collection still has some formatting issues making a high quality keyword extraction a difficult task. While the simple approach of taking all nouns contained in the message text and the English Moby nouns list could clearly be optimised, we believe that the so obtained quality is sufficient for this thesis.

In order to test all implemented schemes with document collections of different sizes, the keyword extraction is done for all news in the Reuters collection. However, only a limited amount is used for the document collections. As shown in Table 5.1, the tests have been performed with document collections containing between 1000 and 10000 documents, where all documents of smaller collections are contained in all larger collections, i.e., Reuters1000 is part of Reuters2000 and so on.

In the rest of this thesis, the Reuters1000, Reuters5000 and Reuters10000 representing a small, medium and large collection are used for analysing the search performances. For the remaining document collections, all details such as queries and search performances are listed in the appendix (see Appendix B and C).

5.3.2. Queries

To cover a wide spectrum of practical situations, single and multiple keyword queries resulting in all kind of output sizes have been tested. Since finding multiple keyword queries for smaller document collections is a time-consuming task and the OXT scheme is the only tested one supporting non-naive multi-keyword queries, the maximum query length is restricted to three keywords.

Collection Name	Number of Documents	Number of Keywords	Size [MB]
Reuters1000	1000	2273	0,845
Reuters2000	2000	3176	1,663
Reuters3000	3000	3853	2,470
Reuters4000	4000	4336	3,318
Reuters5000	5000	4794	4,278
Reuters6000	6000	5179	5,124
Reuters7000	7000	5503	5,955
Reuters8000	8000	5835	6,909
Reuters9000	9000	6126	7,812
Reuters10000	10000	6398	8,757

Table 5.1.: Document Collections.

Search Query	Reuters1000	Reuters5000	Reuters10000
government	91	405	889
loss	90	448	887
securities	56	371	758
meeting	53	262	569
industry	47	241	503
countries	33	191	441
payments	31	153	310
proposal	23	132	253
brazil	21	97	183
coffee	17	62	102
shipment	8	46	80
strategy	7	37	83
hardware	2	12	22
cancer	1	6	13
laboratory	0	8	14

Table 5.2.: Document Matches for Single Keyword Queries.

Search Query	Reuters1000		Reuters5000		Reuters10000	
loss, year	54	(90,269)	261	(448,1376)	509	(887,2759)
government, year	44	(91,269)	182	(405,1376)	392	(889,2759)
industry, year	27	(47,269)	117	(241,1376)	251	(503,2759)
offering, securities	14	(43,56)	70	(212,371)	144	(453,758)
meeting, agreement	12	(53,85)	54	(262,402)	121	(569,846)
brazil, countries	8	(21,33)	30	(97,191)	59	(183,441)
crisis, payments	8	(13,31)	23	(50,153)	43	(109,310)
petroleum, energy	1	(18,12)	22	(89,110)	41	(195,252)
law, proposal	1	(11,23)	8	(89,132)	10	(170,253)
strategy, failure	1	(7,8)	4	(37,39)	5	(83,75)

Table 5.3.: Document Matches for Two Keyword Queries.

As shown in Table 5.2, several single keyword queries have been selected. Quite obviously, the aim behind the keyword selection is providing a uniform distribution of the output sizes.

Considering the two keyword queries shown in Table 5.3, where the values in parentheses define the number of documents found for the single keywords, it has to be noted that the keywords within the query are ordered by frequency, i.e., the least frequent keyword is the first one. In two of the queries (”petroleum, energy” and ”strategy, failure”) the least frequent keyword changes depending on the document collection, but the keyword order is fixed in a way that the order is optimal for the majority of collections. The ordering of keywords is also applied to the three keyword queries shown in Table 5.4

Search Query	Reuters1000		Reuters5000		Reuters10000	
growth, government, year	12	(47,91,269)	39	(207,405,1376)	75	(446,889,2759)
investment, government,year	10	(63,91,269)	26	(262,405,1376)	64	(547,889,2759)
creditor, payments,debt	9	(16,31,74)	17	(44,153,390)	33	(81,310,774)
quotas, meeting,agreement	8	(16,53,85)	12	(51,262,402)	21	(78,569,846)
brazil, debt, president	7	(21,74,64)	19	(97,390,375)	36	(183,774,783)
growth, investment, government	6	(47,63,91)	20	(207,262,405)	37	(446,547,889)
volume, trading, year	5	(12,48,269)	12	(73,241,1376)	22	(142,511,2759)
outlook, economy, growth	4	(12,32,47)	11	(53,130,207)	21	(94,280,446)
dividend, assets, earnings	3	(32,28,35)	3	(165,158,201)	5	(346,368,429)
dollar, system, industry	2	(34,51,47)	3	(155,225,241)	3	(375,449,503)

Table 5.4.: Document Matches for Three Keyword Queries.

5.3.3. Optimal Keyword Order

Due to using an optimal keyword order within each query, the question arises how the keyword order can be computed in practice. Considering the general case of documents containing free text, using an existing frequency list (i.e., a list of all words and their frequency) for the respective language seems to be suitable. Since the vocabulary is highly depended on the context, this approach is only practical if a specific frequency list is available. Consequently, an additional data structure storing information about the keyword distribution is needed.

During the index generation, the client can easily obtain the frequency of each keyword in the document collection. Following the ideas from [Cas+13], several space-efficient data structures (for example Bloom filters) can be used to store sets of keyword with different frequencies. For a better understanding, an example using the Reuters10000 collection, whose keyword frequency and Bloom filter sizes (with a desired false positive rate of $0,01$) are shown in Table 5.5, is presented.

Document Matches	Keywords	Bloom Filter Size [kB]
≥ 500	23	$0,02$
≥ 200	101	$0,13$
≥ 100	226	$0,28$
≥ 50	423	$0,53$
≥ 25	762	$0,96$
≥ 10	1498	$1,89$

Table 5.5.: Keyword Frequency for Reuters10000.

In the Reuters10000 collection, about 77% of the keywords are contained in less than 10 documents. By using a few Bloom filters storing the sets of keywords contained in at least 10, 25, 50, 100, 200 and 500 documents, a total space of less than 4 kilobytes is needed. Consequently, the frequency of a keyword can be estimated efficiently and using the nearly optimal keyword order in the experiments seems to be reasonable.

5.3.4. Environment

All test have been performed using the same physical device whose hardware specification and software configuration is shown in Table 5.6.

Item	Details
Processor	Intel® Xeon® X5690 @ 3.47GHz
Memory	192 GB RAM
Operating System	Debian GNU/Linux Stretch (testing status)
Kernel Version	4.6.4-1 (2016-07-18)
Java™ Version	1.8.0_92 (64-Bit Server VM)

Table 5.6.: Hardware and Software Configuration.

Chapter 6

Experimental Results

In this chapter, the experimental results are discussed. For all tests and schemes, the security parameter k has been set to 128, i.e., a key length of 128 bits is used.

6.1. BuildIndex Performances and Index Sizes

The results of all `BuildIndex` algorithm represent the average of 50 executions. As can be seen in Table 6.1 and Figure 6.1, there are tremendous differences in the schemes index sizes and building performances.

Secure Index Scheme: By using space efficient Bloom filters for the index construction, the forward indices are of small size resulting in a space efficiency inverse proportional to the number of documents. Since the few[1] PRF computations are the only time relevant operation, forward indices can be calculated extremely efficient resulting in an outstanding overall index generation performance.

PPSED Scheme: This scheme maps each keyword in Δ to a bit in the forward index, producing an overall index being a lot smaller compared to the other ones. While the pseudorandom mapping can be pre-calculated efficiently, all keywords have to be tested in all documents and an individual masking has to be calculated. Due to the extensive amount of operations, the forward index calculation is time-consuming resulting in a slow overall index generation performance proportional to the number of documents.

SSE-1 Scheme: Starting with array A, both the index size and generation time depend on the number of nodes, where the number of nodes depends on the total number of keywords in the document collection. In contrary, for the look-up table T the size of the dictionary is important. Since the used document collections are structurally related, both the index size and generation time show the expected linear dependency of the document collection size. In comparison to other schemes, it has to be noted that the SSE-1 construction hides q' and the number of keywords contained in $\mathcal{D}$ by blinding A and T. Without this blinding, the index generation time would be significantly faster (but still slower than the secure index scheme) and the index sizes would be a lot smaller (but still larger than the secure index scheme).

[1] For the desired false positive rate of $0,01$ used in the test setting, the number of hash functions $r = \lceil -\log_2(0,01) \rceil = 7$.

KRB Scheme: In the KRB scheme, each node contains two hash tables both having an encrypted entry for all keywords in Δ. Since the hash table key needs k bits and the value is of block size length, each hash table needs $|\Delta| \cdot 128 \cdot 128$ bits in the test setting[2]. Since the total number of nodes is $2n-1$, the KRB scheme produces indices of tremendous size. Moreover, the index generation involves a lot of encryptions resulting in a slow index generation performance.

OXT Scheme: The index construction stores two small size entries for each document-keyword combination without the need to add any blinding values. Consequently, the index sizes are much smaller compared to other inverted index schemes and scale very well with increasing amount of documents. Since the total number of involved computations is very low, the index can be build very efficiently, even though time consuming asymmetric cryptography is involved.

Document Collection	Secure Index	PPSED	SSE-1	KRB	OXT
Reuters1000	2,915	1,254	117,116	236,533	3,191
Reuters2000	8,103	2,507	230,456	661,218	6,331
Reuters3000	14,714	3,761	342,235	1203,196	9,420
Reuters4000	22,053	5,015	459,981	1805,231	12,602
Reuters5000	30,454	6,269	573,330	2495,532	16,279
Reuters6000	39,451	7,523	711,434	3236,062	19,517
Reuters7000	48,884	8,777	826,690	4011,530	22,672
Reuters8000	59,215	10,031	959,989	4861,622	26,279
Reuters9000	69,908	11,285	1086,270	5742,012	29,748
Reuters10000	81,114	12,539	1217,668	6663,235	33,323

Table 6.1.: Index Size [MB].

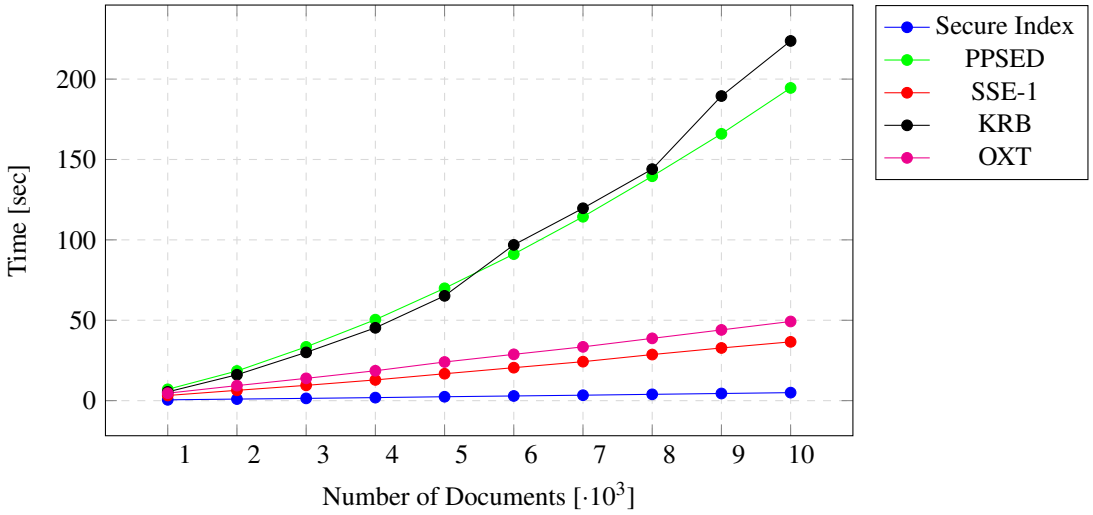


Figure 6.1.: BuildIndex Performances.

[2] If stored to disk, for each node it is sufficient to store each hash table key just once since the hash tables differ only in the values. This optimisation has been applied to the indices shown in Table 6.1

6.2. Single Keyword Searches

To obtain meaningful results, all experiments represent the average of 100 executions. Obviously, this test procedure has also been applied to the later discussed multiple keyword queries.

As can be seen in Figure 6.2, there already exist search time differences for single keyword queries on a rather small document collection. With increasing amount of documents resulting in search performances shown in Figures 6.3 and 6.4, the characteristics of the schemes become more evident.

First of all, the search performance of the forward index-based SSE schemes (secure index and PPSED scheme) is proportional to the number of documents, as the example of the PPSED scheme taking about 20 ms for Reuters5000 and 40 ms for Reuters10000 illustrates.

In contrary, the search performance of the inverted index-based SSE schemes (SSE-1, KRB and OXT scheme) is proportional to the number of documents containing the keyword. Since the output sizes of the search queries are almost proportional to the number of documents, the search performances of all schemes scale quite similar.

Secure Index Scheme: For each document, a fixed amount of HMAC-SHA256 computations are required to compute the codeword. By reason of the non-occurrence of false positives in the applied test setting, the amount of bits that have to be tested decreases if a codeword is not contained in the Bloom filter. However, the HMAC-SHA256 computations are the far more time consuming task, resulting in a search performance being almost constant in the number of documents. Since the number of computations is much higher compared to other schemes, the inefficient search performance can not compete with any of them.

PPSED Scheme: Since each keyword is related to one index bit, the search performance solely depends on the number of documents resulting in almost constant search times for each collection. Consequently, the forward index construction limits the scheme's usability for larger document collections.

SSE-1 Scheme: The search performance clearly depends on the number of found documents, since for each one an encrypted node in the hidden linked list has to be processed. Since iterating through the linked list is time expensive, the subsequent schemes outperform the SSE-1 scheme if more nodes have to be processed, i.e., more documents are found.

KRB Scheme: Depending on the number of found documents, only a part of the tree has to be searched. Since only one entry in each relevant node has to be processed, the total number of computations is lower compared to other schemes resulting in a superior search performance.

OXT Scheme: In the case of single keyword searches, the X-Set does not need to be accessed but only the T-Set is used. Consequently, time consuming elliptic curve computations are avoided, which results in an excellent search performance.

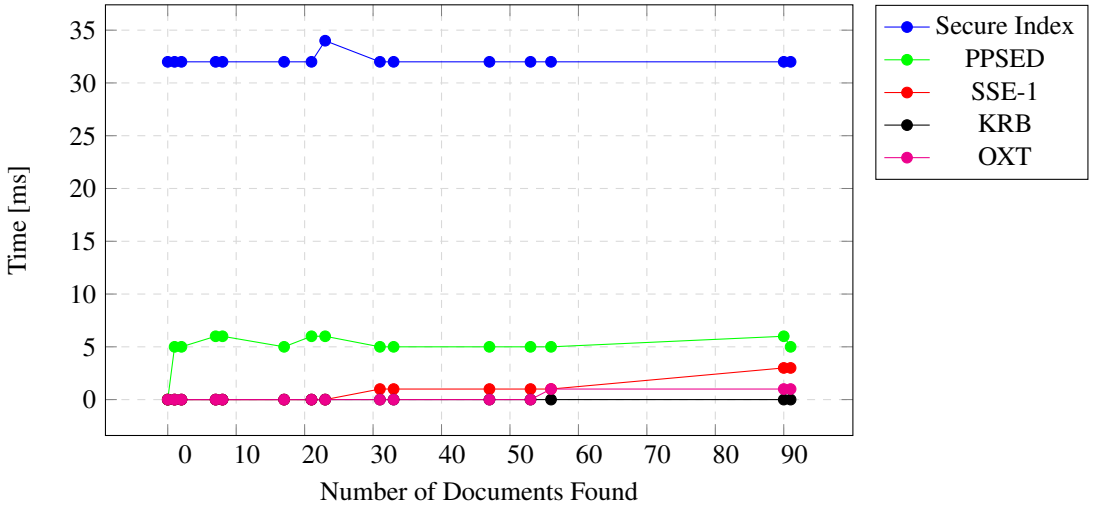


Figure 6.2.: Search Performance Reuters1000: Single Keyword Queries.

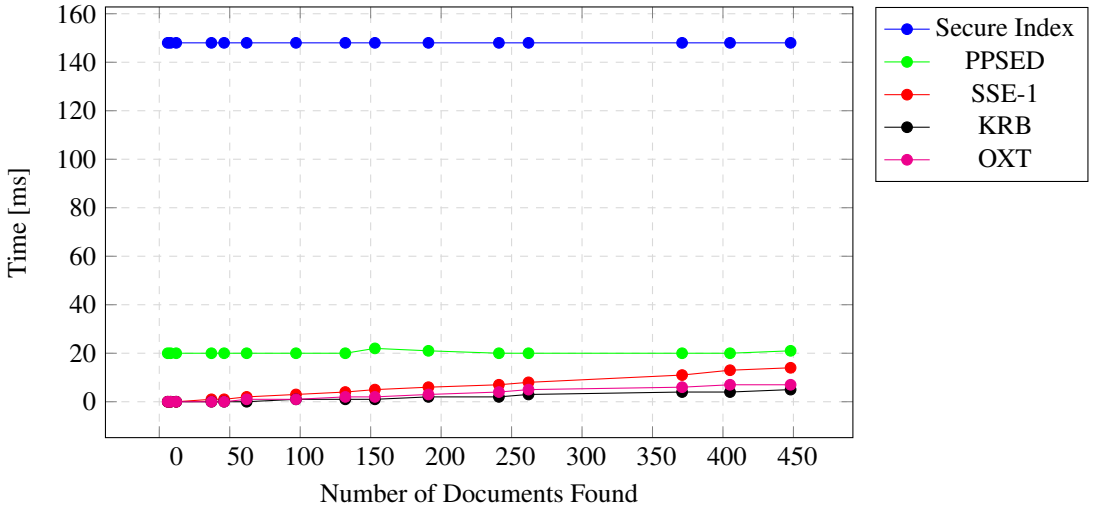


Figure 6.3.: Search Performance Reuters5000: Single Keyword Queries.

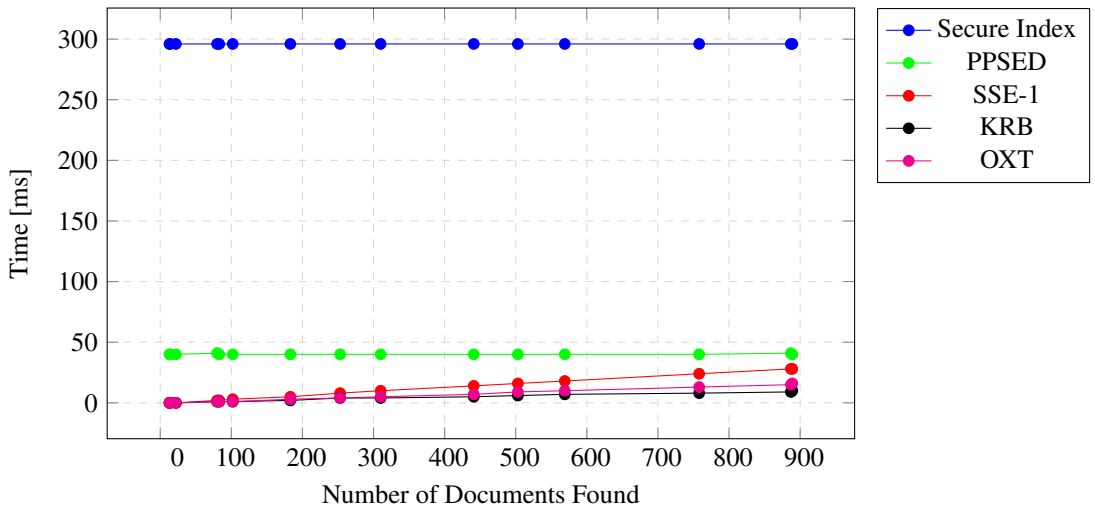


Figure 6.4.: Search Performance Reuters10000: Single Keyword Queries.

6.3. Two Keyword Searches

Since forward index-based SSE schemes process each document individually, they reduce the amount of keyword searches in multiple keyword queries because there is no need to continue searching the document once a keyword is not contained. Therefore, for both forward index-based schemes, the search performance is based on excluding documents quickly which is improved by using the optimal keyword order within each query. As can be seen by the comparison of Figure 6.7 and Figure 6.8 (having a logarithmic scale), if the keywords are in the suboptimal order, i.e., the most frequent keyword is the first keyword in the query, the search performances of the forward index-based schemes and the OXT scheme are affected. Following this finding, all tests are done using optimal ordered keyword queries.

In the case of conventional inverted index-based schemes (SSE-1 and KRB scheme), the index constructions do not allow reducing the document collection efficiently. Consequently, two separate single keyword queries are executed and the intersection of the result sets is calculated optimally, i.e., the smaller result set is fixed for the computation.

Secure Index Scheme: Since a fixed amount of HMAC-SHA256 operations are required to compute the codeword of a keyword, the scheme benefits from optimal keyword order since total number of HMAC-SHA256 computations can be reduced. While the optimal ordering affects the practical performance for queries containing a highly frequent keyword, the total amount of operations is much higher compared to other schemes resulting in an inefficient search performance regardless of the keyword order.

PPSED Scheme: Quite similar to the single keyword case, the PPSED scheme achieves almost constant search times depending mostly on the number of documents. While the number of keyword tests is reduced as in the secure index scheme, the performance impacts are insignificant since the number of bits that have to be checked is reduced by at most one. Consequently, the keyword order is not of a major concern. With increasing number of documents, the difference to the fast KRB scheme is reduced, but in comparison to the KRB scheme providing the adaptive indistinguishability notion (IND-CKA2), the PPSED leaks more information and achieves the weaker non-adaptive indistinguishability notion (IND-CKA1).

SSE-1 Scheme: Due to treating a multiple keyword query as multiple single keyword queries, the same reasoning as in the single keyword case applies. Since the three queries with the most overall documents found include the very frequent keyword ”year” included in about 27%[3] of all documents, these queries result in a moderate search performance.

KRB Scheme: Similar to the SSE-1 scheme, the reasoning of the single keyword queries can be applied. While the search time of each single keyword query increases proportional to the number of found documents, the overall performance is still excellent due to the outstanding single keyword search times.

OXT Scheme: The OXT scheme uses the first keyword, which is expected to be the least frequent one, to search for possible document identifiers in the T-Set whose containment of the second keyword is then checked using the X-Set. Processing the X-Set is the performance limiting factor in practice, since computations on the NIST P-224 elliptic curve are far more time consuming compared to the simple HMAC-SHA256 operations involved in the T-Set processing. Consequently, the overall search time highly correlates to the X-Set involvement, which itself depends on the output size of the T-Set search. Therefore, the search performance depends on the number of documents found for the first keyword, but is independent on number of documents found for both keywords.

To provide an example, the queries q_1 =”meeting, agreement” and q_2 =”industry, year” are considered for Reuters10000. While executing q_1 returns 121 documents and q_2 outputs 251 documents, q_1 needs 514 ms where q_2 takes only 453 ms. Since the keyword ”meeting” is contained in 569 documents and ”industry” in 503, it is easy to see that the search time solely depends on the first keyword. Therefore, the second keyword and the number of found documents for the query are entirely irrelevant regarding the search time.

As final note toward the OXT multiple keyword performance, it has to be emphasised that the comparison with schemes leaking all single keyword results is delusive. If the OXT scheme would leak all single keyword results by using only the T-Set, the search performance would be similar to the KRB scheme, as the single keyword results have proven.

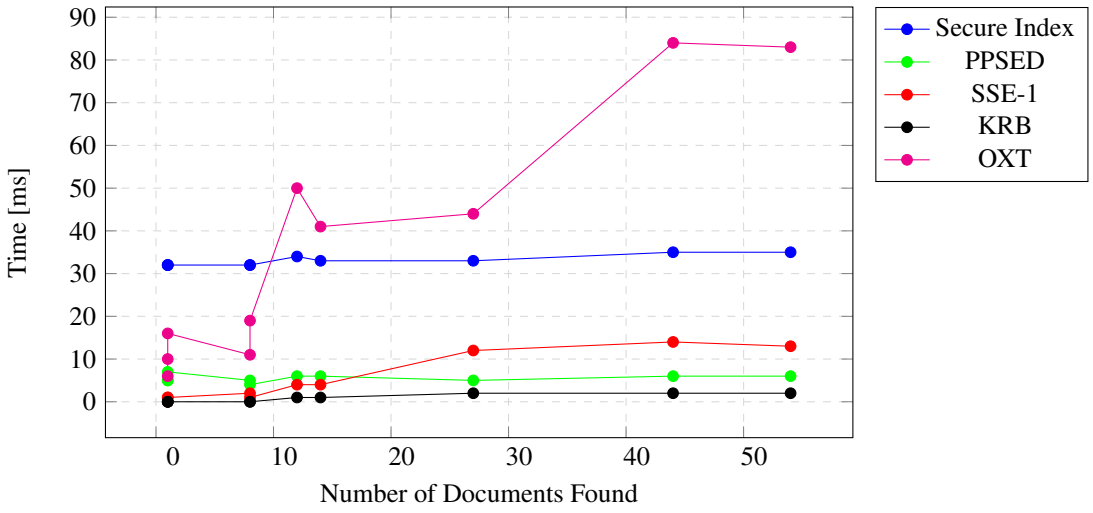


Figure 6.5.: Search Performance Reuters1000: Two Keywords per Query.

[3] 26,9% in Reuters1000; 27,52% in Reuters5000 and 27,59% in Reuters10000

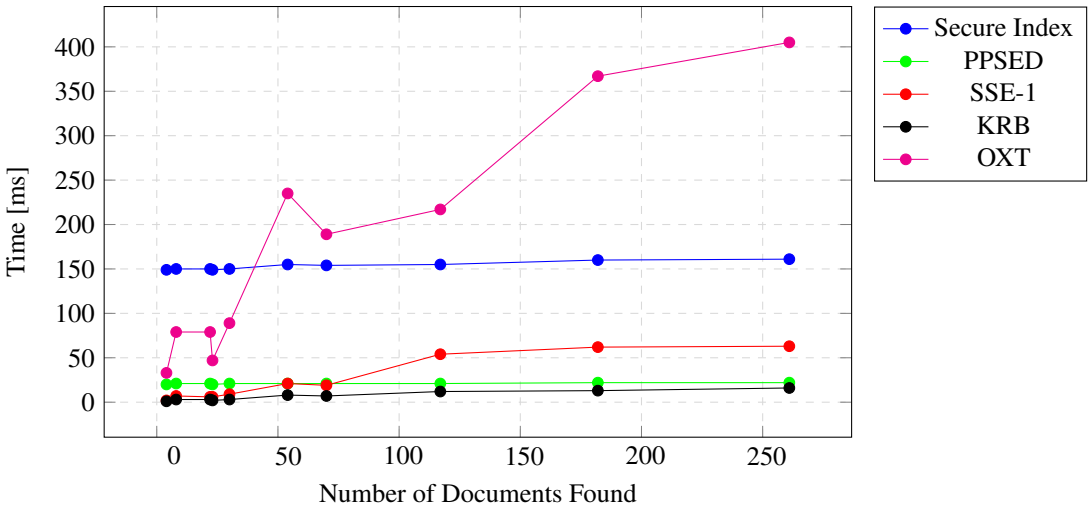


Figure 6.6.: Search Performance Reuters5000: Two Keywords per Query.

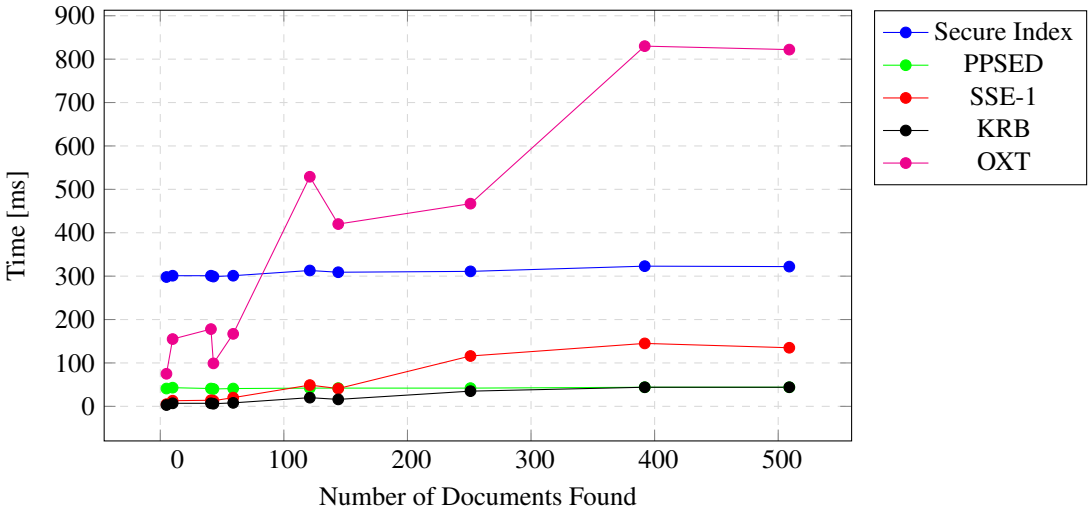


Figure 6.7.: Search Performance Reuters10000: Two Keywords per Query.

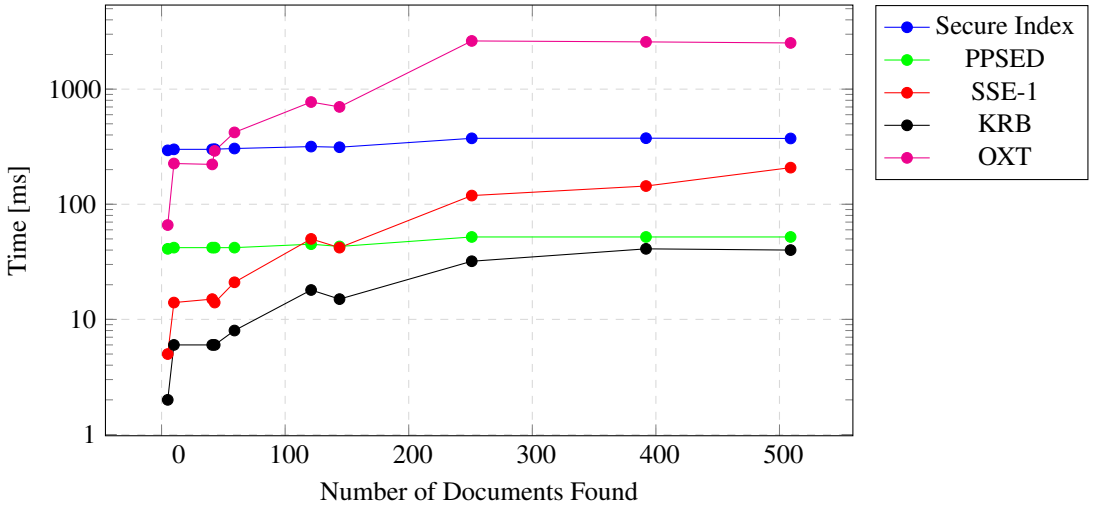


Figure 6.8.: Search Performance Reuters10000: Two Keywords per Query with Most Frequent Keyword First.

6.4. Three Keyword Searches

As can be seen in Figures 6.9, 6.10 and 6.11, the outcome of the three keyword queries shows expected performance results with respect to the previously discussed two keyword queries.

While the forward index-based schemes already benefited from reducing the number of actual keyword searches in the case of two keyword queries, this reduction becomes more advantageous the more keywords are contained in the query. Consequently, the performance of the PPSED scheme and the KRB scheme converges, but the KRB is still able to outperform all schemes regardless of the number of found documents. If the lengths of the query would be further extended, the PPSED scheme would achieve better multiple keyword search performance than the KRB scheme at some point. Regarding the keyword order within queries, it can be seen in Figure 6.12 (having a logarithmic scale) that mainly the OXT performance heavily depends even more on the first keyword being the least frequent one than in the two keyword tests.

All in all, the keyword searches need proportionately more time compared to the two keyword searches and the same patterns can be identified.

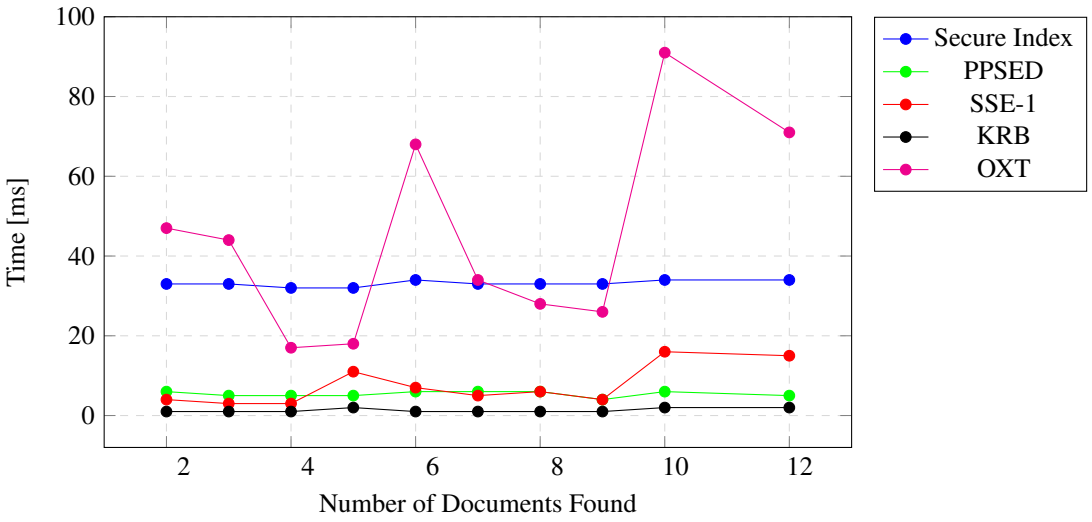


Figure 6.9.: Search Performance Reuters1000: Three Keywords per Query.

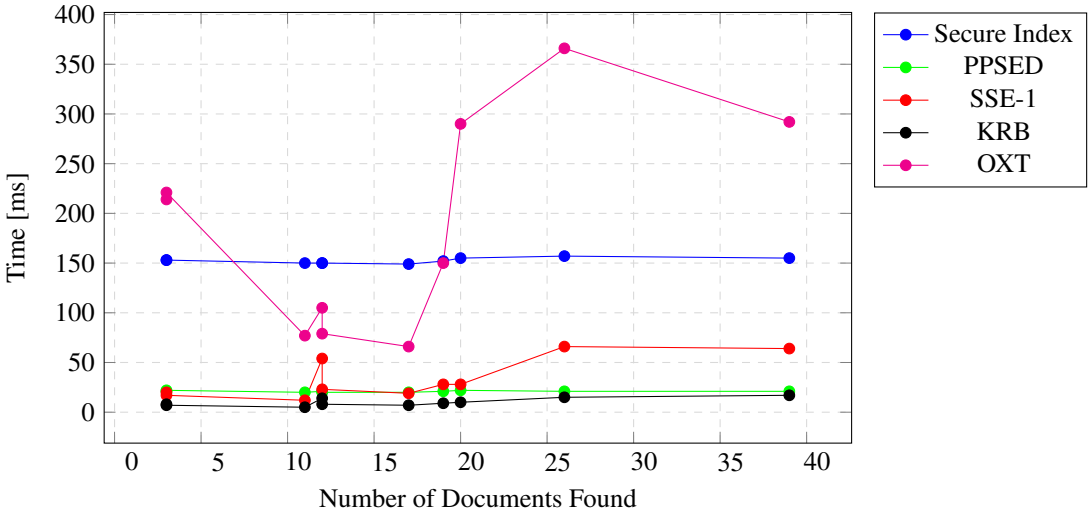


Figure 6.10.: Search Performance Reuters5000: Three Keywords per Query.

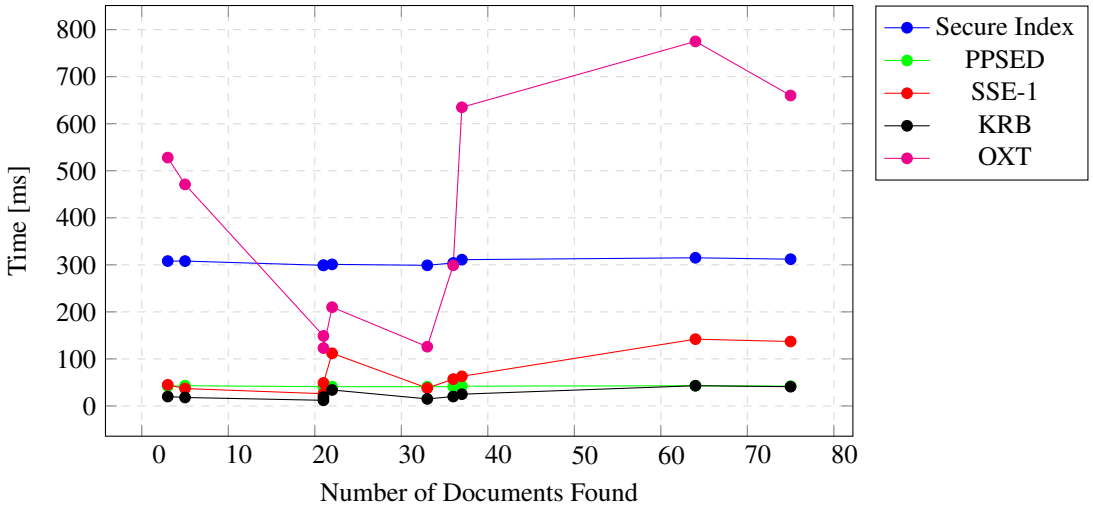


Figure 6.11.: Search Performance Reuters10000: Three Keywords per Query.

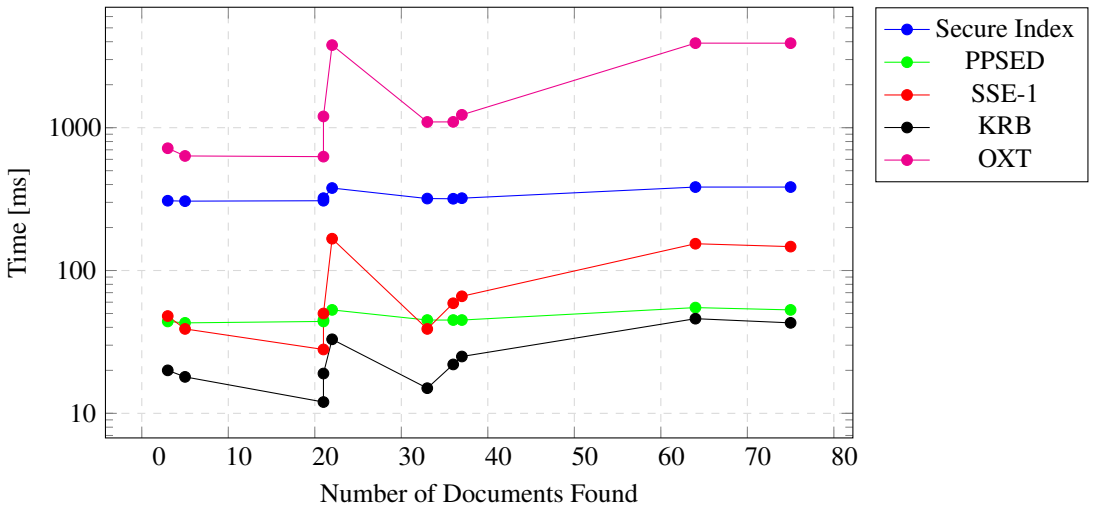


Figure 6.12.: Search Performance Reuters10000: Three Keywords per Query with Most Frequent Keyword First.

Chapter 7

Findings

In this chapter, the findings of the theoretical analysis of the selected Searchable Symmetric Encryption (SSE) schemes and the implementation results will be combined to an overall comparison and some recommendations regarding the use of the schemes will be provided.

7.1. Asymptotic Overview

Based on the findings of the previously presented comprehensive analysis of individual schemes, Table 7.1 shows a comparison of the asymptotic performances, where all the variables are defined in Table 7.2.

Scheme	Security	Updates	BuildIndex	Search	Storage
Secure Index [Goh03]	IND-CKA1	Yes	$O\left(\frac{q \cdot n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$O(n)$
PPSED [CM05]	IND-CKA1	No	$O\left(\frac{d \cdot n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$O(d \cdot n)$
SSE-1 [Cur+06]	IND-CKA1	No	$O\left(\frac{q \cdot n}{p}\right)$	$O(\lvert\mathcal{D}(w)\rvert)$	$O(d+s)$
KRB [KP13]	IND-CKA2	Yes	$O\left(\frac{q \cdot n}{p}\right)$	$O\left(\frac{\lvert\mathcal{D}(w)\rvert \cdot \log n}{p}\right)$	$O(d \cdot n)$
OXT [Cas+13]	IND-CKA2	Yes	$O\left(\frac{q'}{p}\right)$	$O\left(\frac{\lvert\mathcal{D}(w')\rvert}{p}\right)$[1]	$O(q')$

[1] The OXT scheme takes $O\left(\frac{q \cdot \lvert\mathcal{D}(w')\rvert}{p}\right)$ for a query of q keywords. For a better comparison, the single keyword case is shown in the table.

Table 7.1.: Comparison of SSE Schemes.

7.2. Usage Recommendations

In this section, recommendations for certain typical practical SSE requirements are provided. While these recommendations are founded on both the theoretical analysis and implementation results, it has has to

Notation	Meaning
$\mathcal{D}$	document collection
n	number of documents
d	number of entries in the dictionary
q	number of distinct keywords per document
q'	sum over all keywords in $\mathcal{D}$, i.e., $q' = \sum_{w_i \in \Delta'} \lvert\mathcal{D}(w_i)\rvert$
$\lvert\mathcal{D}(w)\rvert$	number of documents containing keyword w
$\lvert\mathcal{D}(w')\rvert$	number of documents containing the least frequent keyword w'
s	size of $\mathcal{D}$ expressed in smallest possible keyword size
p	number of processors (cores)

Table 7.2.: Notation for SSE Schemes.

be noted that they are partly a matter of personal preference.

7.2.1. Search Performance

The requirement that will most likely be the most common in practice is that keyword searches have to be as efficient as possible without leaking critical information. While all presented schemes provide reasonable security guarantees and acceptable search performances, the keyword red-black (KRB) scheme outperforms all other ones in almost every test case done in this thesis. Since the search process could be parallelised to further improve the performance, the KRB scheme is recommended in settings where optimal search efficiency is needed.

If the significant space requirements of the KRB scheme should be a deal-breaker, the SSE-1 scheme, achieving asymptotically optimal search time proportional to the number of found documents for single keyword queries, has to be considered. The performance related issue with the SSE-1 scheme is that it does not process queries containing a high frequent keyword as efficient as some other schemes, a drawback which comes into place mainly if such a keyword is part of a multiple keyword query. Additionally, it is not possible to parallelise the search process of the SSE-1 scheme.

If it is known that most of the time multiple keywords queries are executed and the KRB scheme can not be applied, the Privacy Preserving Keyword Searches on Remote Encrypted Data (PPSED) scheme benefiting from its forward index-based construction is a good choice in such settings. If only single keyword queries are applied, the Oblivious Cross-Tags (OXT) scheme provides almost equal search performance as the KRB scheme since only the T-Set but not the X-Set has to be processed. If all types of queries are equally likely to appear and the available resources require a sequential execution of search queries, the SSE-1 scheme seems to be the best all-round alternative to the KRB scheme .

7.2.2. Security

In search for a SSE scheme providing the best possible security while still being practical, the strong adaptive security (IND-CKA2) has to be achieved. Therefore, the KRB scheme and the OXT scheme providing IND-CKA2 security are suitable choices.

As a recall of the IND-CKA2 definition, the leaked information contains the access pattern covering the result of a search request. Since the OXT scheme is able to non-naively handle arbitrary multiple keyword queries, it only leaks the query's result and not all single keyword search results as the KRB

scheme does. While the OXT scheme achieves the best security guarantees, its search performance for multiple keyword queries can not compete with the other presented SSE schemes in practice due to the involvement of asymmetric cryptography. Consequently, it is advisable to use the OXT scheme if minimising the information leakage is the most important issue, but increased search times for multiple keyword queries can be tolerated.

If it is known that only single keyword queries are executed, the OXT and KRB schemes provide the same security level. In this case, using the KRB scheme instead of the OXT scheme would provide slightly increased search performance at the drawback of immense storage needs. Therefore, the OXT scheme is recommended in all settings aiming to prevent information leakage.

7.2.3. Space Efficiency

In a setting where only a very limited amount of storage space is available, the PPSED scheme, the secure index scheme and the OXT scheme are possible candidates as the implementation results reveal.

If space efficiency has to be achieved regardless of other performance indicators, the PPSED scheme should be chosen since its indices are less than half the size of the next smaller one for all tested document collections. One of the drawbacks of the PPSED scheme is that building the indices takes a lot more time compared to the other candidate schemes, especially compared to the secure index scheme. Additionally, it has to be recalled that the PPSED scheme utilises a dictionary, a characteristic that might precludes using this scheme.

With the condition that using a dictionary is not possible, the OXT scheme achieves the best space efficiency of the remaining schemes. While the OXT scheme's storage space is linear in the total number of keywords in the document collection, the secure index requires space linear in the number of documents. As indicated by the implementation results, the OXT scheme needs less space for medium and large document collections (Reuters4000 upwards), but the secure index scheme produces smaller indices if the number of documents is rather low (until Reuters3000). Consequently, the OXT scheme is the better choice for space efficiency considering arbitrary document collections, but building the OXT index takes a lot more time compared to the secure index scheme. Therefore, if low storage costs are desirable but not at the expense of the index generation performance, the secure index scheme provides a reasonable trade-off of all index related requirements.

7.2.4. Index Generation Efficiency

Assuming a setting where many clients are very limited in their computational power, the client's operations being the index and trapdoor generation have to be as efficient as possible. As expected, all schemes are able to compute the trapdoors efficiently making the index generation performance the primary selection criteria. Consequently, the secure index scheme is the best choice for such an environment.

If an inverted index-based scheme is desired, the SSE-1 and OXT scheme have to be considered. Even though the SSE-1 scheme is able to build the indices more efficiently, using the OXT is recommended since it produces much smaller indices and the clients will most likely be limited in memory as well.

7.2.5. Efficient Updates

In a setting where updates of the document collection occur frequently, a dynamic SSE scheme should be used. Depending on the specific type and complexity of the update, the secure index scheme, the KRB scheme and the OXT scheme have to be considered.

At first, a distinction has to be made whether most of the time existing documents are updated or merely new document are added. In the much simpler latter case, the secure index scheme provides the advantage that due to its forward index construction, no update has to be done at all. Since the secure index scheme outperforms all other tested schemes in the index building process, adding new documents can be done very efficiently. While the secure index scheme also supports efficient document updates in form of adding keywords, it is not possible to delete keywords efficiently requiring a complete index rebuilding.

Considering the KRB scheme supporting arbitrary updates, it has been argued (see Section 4.4.2) why the update algorithm needs additional knowledge of the initial data vectors that could be provided by encrypting the initial KRB tree. While this solution is asymptotically efficient, it has to be considered that building the index takes a lot of time compared to other candidate schemes. Consequently, if a lot of documents have to be updated (regardless of the number of updated keywords within each document), the update performance converges towards the index generation performance resulting in updates being slower than the index generation of other schemes. Therefore, the KRB scheme provides reasonable update performance for all kind of updates, as long as it affects only a limited amount of documents.

Regarding the OXT scheme, again all types of updates are supported. Similar to the secure index scheme, adding documents can be done very efficiently, but deleting documents is the more expensive operation. While the other schemes' update performances scale with the number of involved documents, the OXT update times depend on the number of contained keywords. In the worst case, i.e., deleting a document that contains all keywords appearing in other documents, the update time takes about half the time of the index generation since half of the T-Set has to be rebuild if the lists have been randomised. In situations where only a very limited amount of keywords is contained in an update, the OXT scheme is able to perform updates efficiently.

All in all, the secure index scheme is recommended in settings where updates have to be performed efficiently, especially if it is known that deleting keyword operations rarely occur. If large document collections with frequent arbitrary updates have to be processed, the expected update performance will converge towards the index generation performance, which would still outperform all presented static schemes and the KRB scheme. If updates typically involve only a few keywords, using the OXT scheme would grant the benefits of achieving a stronger security model and smaller index size for larger document collections.

7.2.6. All-round Performance

In a situation where none of the above discussed requirements is significantly outstanding or everything is equally important, a scheme achieving a reasonable performance in all aspects without having major drawbacks is desirable. Combining the findings of the asymptotic overview and the implementation results, it can be seen that the SSE-1 provides reasonable overall performance with the sequential search process being the only substantial weakness.

While the SSE-1 indices are larger than most other ones, they are nowhere close to the huge KRB ones and can be generated very efficiently. As discussed before, only the secure index scheme is able to generate indices faster. Considering the search performance, the SSE-1 scheme achieves the third best efficiency for both single and multiple keyword queries. While for all specific settings – for example small document collections with single keyword searches or large document collections with multiple keyword searches – schemes providing better search performance do exist, the SSE-1 scheme is able to keep up in all scenarios and has no major lack of search efficiency regardless of the situation.

Chapter 8

Conclusion and Further Work

In this thesis, the importance and usability of Searchable Symmetric Encryption (SSE) has been motivated on the basis of a fundamental theoretical analysis and practical performance evaluation of selected SSE schemes.

As the analysis has shown, a lot of different construction methods exist, varying in achieved tradeoff between security and performance. The comparison of schemes revealed that the forward index-based construction apply simpler construction methods, but leak more information resulting in non-adaptive security (IND-CKA1). The inverted index-based construction are typically more complex, but often hide more information achieving the stronger adaptive security (IND-CKA2).

According to the practical performance evaluation, inverted index-based scheme are superior in most practically relevant situations. Even though the forward index constructions can not compete with the fasted inverted index ones, it is fair to say that their performance is good enough for using them in practice. Nevertheless, depending on the concrete usage scenario, in most situations using a state-of-the-art inverted index-based scheme is advisable due to the better security guarantees and practical performance.

Since this thesis provided an overview about searchable encryption with focus on SSE, it layed a starting point for possible future research. In the following, some of the possible challenges that could investigated in the future are discussed.

Evaluation of Parallelised Performance: All performance tests have been sequentially executed. As important further work, the implementation could be adapted to allow a parallelised index generation and keyword search.

Evaluation of Update Performance: So far, all schemes have been implemented without update capabilities. For those supporting efficient updates, the update algorithm could be implemented. If a scheme does not support updates, the trivial solution of rebuilding the index could be used to get a comparison value for the performance evaluation.

Extension of Tested Schemes: While the scheme selection was motivated by providing an overview about different fundamental constructions, it could be interesting to include other schemes, for example some of the schemes mentioned in the introduction. Obviously, the SSE-2 scheme could be added for a practical comparison to the already implemented SSE-1 scheme.

Evaluation of Other Document Collections: Even though different document collections in terms of sizes and contained keywords were tested, the general structure of all collections was related. As

possible extension of the test setting, using other document collections which have a different ratio of keywords to number of documents, very small or large size and a different distribution of keywords could be tested.

Evaluation on Devices with Limited Resources: All performance tests have been done on the same physical device provided with a reasonably up-to-date processor and a lot of memory. While it is foreseeable that limiting the memory would exclude the usage of some schemes for larger document collections, it could be investigated how the index generation performance relates to the powerfulness of the processor.

Extension of Search Queries: Due to time-consumption of finding meaningful multiple keyword queries, the search query lengths have been limited to three keywords. As obvious further work, tests with longer queries could be made to emphasise schemes' search behaviour.

Integration into a Cloud Storage Framework: The implementation done during this thesis can be considered as a library for SSE schemes that could be integrated into an existing framework for building secure cloud storages to evaluate the security and performances of cloud storages on a more comprehensive basis.

Bibliography

[Ara+13] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. "Orthogonal Security with Cipherbase". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research*. 2013.

[Bab+12] Steve Babbage et al. "ECRYPT II Yearly Report on Algorithms and Keysizes". In: *European Network of Excellence in Cryptology II, Tech. Rep* (2012).

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. "Deterministic and Efficiently Searchable Encryption". In: *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference*. 2007, pp. 535–552.

[BFP16] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. "Verifiable Dynamic Symmetric Searchable Encryption: Optimality and Forward Security". In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 62.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. "Biclique Cryptanalysis of the Full AES". In: *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security*. 2011, pp. 344–371.

[Ble98] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference*. 1998, pp. 1–12.

[Blo70] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (1970), pp. 422–426.

[Bon+04] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. "Public Key Encryption with Keyword Search". In: *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques*. 2004, pp. 506–522.

[BRS10] Mihir Bellare, Phillip Rogaway, and Terence Spies. "The FFX mode of Operation for Format-Preserving Encryption". In: *NIST submission* (2010).

[BSS08] Joonsang Baek, Reihaneh Safavi-Naini, and Willy Susilo. "Public Key Encryption with Keyword Search Revisited". In: *Computational Science and Its Applications - ICCSA 2008*. 2008, pp. 1249–1259.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. "Functional Encryption: Definitions and Challenges". In: *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011*. 2011, pp. 253–273.

[Cas+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries". In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*. 2013, pp. 353–373.

[Cas+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation". In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014*. 2014.

[Cas+15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. "Leakage-Abuse Attacks Against Searchable Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 668–679.

[Cho+95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. "Private Information Retrieval". In: *36th Annual Symposium on Foundations of Computer Science*. 1995, pp. 41–50.

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. "Privacy Preserving Keyword Searches on Remote Encrypted Data". In: *Applied Cryptography and Network Security, Third International Conference, ACNS 2005*. 2005, pp. 442–455.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)* MIT Press, 2009.

[Cur+06] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*. 2006, pp. 79–88.

[DPR00] Andreas Dandalis, Viktor K. Prasanna, and José D. P. Rolim. "A Comparative Study of Performance of AES Final Candidates Using FPGAs". In: *Cryptographic Hardware and Embedded Systems - CHES 2000*. 2000, pp. 125–140.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*. 2009, pp. 169–178.

[GO96] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *J. ACM* 43.3 (1996), pp. 431–473.

[Goh03] Eu-Jin Goh. "Secure Indexes". In: *IACR Cryptology ePrint Archive* 2003 (2003), p. 216.

[Hal+98] Chris Hall, David Wagner, John Kelsey, and Bruce Schneier. "Building PRFs from PRPs". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference*. 1998, pp. 370–389.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., 2004, p. 312.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation". In: *19th Annual Network and Distributed System Security Symposium, NDSS 2012*. 2012.

[Ish+16] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. "Private Large-Scale Databases with Distributed Searchable Symmetric Encryption". In: *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016*. 2016, pp. 90–107.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2014.

[Knu97] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. 3rd ed. Pearson Education, 1997.

[Kob87] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of Computation* 48 (1987).

[KP13] Seny Kamara and Charalampos Papamanthou. "Parallel and Dynamic Searchable Symmetric Encryption". In: *Financial Cryptography and Data Security - 17th International Conference, FC 2013*. 2013, pp. 258–274.

[KPR11] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. *CS2: A Searchable Cryptographic Cloud Storage System*. Tech. rep. May 2011.

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. "Dynamic searchable symmetric encryption". In: *The ACM Conference on Computer and Communications Security, CCS'12*. 2012, pp. 965–976.

[LM10] M. Lochter and J. Merkle. *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*. RFC 5639 (Informational). Internet Engineering Task Force, Mar. 2010.

[Mao03] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.

[Mil85] Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology - CRYPTO '85*. 1985, pp. 417–426.

[MOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[Nat01a] National Institute of Standards and Technology. *FIPS 197: Specification for the Advanced Encryption Standard (AES)*. Nov. 2001.

[Nat01b] National Institute of Standards and Technology. *SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. Dec. 2001.

[Nat08] National Institute of Standards and Technology. *SP 800-123: Guide to General Server Security*. July 2008.

[Nat10] National Institute of Standards and Technology. *SP 800-132: Recommendation for Password-Based Key Derivation - Part 1: Storage Applications*. Dec. 2010.

[Nat13] National Institute of Standards and Technology. *FIPS 186-4: Digital Signature Standard (DSS)*. July 2013.

[Nat16] National Institute of Standards and Technology. *SP 800-57 Part 1 Rev. 4: Recommendation for Key Management - Part 1: General*. Vol. Revision 4. Jan. 2016.

[Nav15] Muhammad Naveed. "The Fallacy of Composition of Oblivious RAM and Searchable Encryption". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 668.

[NPG14] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. "Dynamic Searchable Encryption via Blind Storage". In: *2014 IEEE Symposium on Security and Privacy, SP 2014*. 2014, pp. 639–654.

[Oga+13] Wakaha Ogata, Keita Koiwa, Akira Kanaoka, and Shin'ichiro Matsuo. "Toward Practical Searchable Symmetric Encryption". In: *Advances in Information and Computer Security - 8th International Workshop on Security, IWSEC 2013*. 2013, pp. 151–167.

[Pop+11] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. "CryptDB: protecting confidentiality with encrypted query processing". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011*. 2011, pp. 85–100.

[Pre93] Bart Preneel. "Analysis and Design of Cryptographic Hash Functions". PhD Thesis. Katholieke Universiteit Leuven, 1993.

[RMÖ15] Cédric Van Rompay, Refik Molva, and Melek Önen. "Multi-user Searchable Encryption in the Cloud". In: *Information Security - 18th International Conference, ISC 2015*. 2015, pp. 299–316.

[Sch+99] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *Performance comparison of the AES submissions*. 1999.

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. "Practical Dynamic Searchable Encryption with Small Leakage". In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014*. 2014.

[SWP00] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. "Practical Techniques for Searches on Encrypted Data". In: *2000 IEEE Symposium on Security and Privacy*. 2000, pp. 44–55.

[ZKP16] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption". In: *25th USENIX Security Symposium, USENIX Security 16*. 2016, pp. 707–720.

Appendix A

List of Acronyms

ABE	Attribute-Based Encryption
AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CCA1	Non-adaptive chosen-chiphertext attack
CCA2	Adaptive chosen-chiphertext attack
CFB	Cipher Feedback
CPA	Chosen-plaintext attack
CTR	Counter
DES	Data Encryption Standard
ECB	Electronic Codebook
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
FE	Functional Encryption
FHE	Fully Homomorphic Encryption
FPE	Format-Preserving Encryption
HMAC	Keyed-hash message authentication code
IAIK-JCE	IAIK Provider for the Java Cryptography Extension
IBE	Identity-Based Encryption
IND-CCA1	Indistinguishability under non-adaptive chosen-chiphertext Attack
IND-CCA2	Indistinguishability under adaptive chosen-chiphertext Attack
IND-CKA	Indistinguishability against an adaptive chosen-keyword attack
IND-CKA1	Non-adaptive indistinguishability for Searchable Symmetric Encryption
IND-CKA2	Adaptive indistinguishability for Searchable Symmetric Encryption
IND-CPA	Indistinguishability under chosen-plaintext attack
IND1-CKA	Semantic security against an adaptive chosen-keyword attack

IND2-CKA	Semantic security against an adaptive chosen-keyword attack
IV	Initialisation vector
KRB	Keyword red-black
NIST	National Institute of Standards and Technology
OFB	Output Feedback
OPE	Order-Preserving Encryption
ORAM	Oblivious Random Access Machine
OXT	Oblivious Cross-Tags
PBKDF2	Password-Based Key Derivation Function 2
PCPA	Pseudorandomness against chosen-plaintext attack
PEKS	Public-Key Encryption with Keyword Search
PIR	Private Information Retrieval
PPE	Property-Preserving Encryption
PPSED	Privacy Preserving Keyword Searches on Remote Encrypted Data
PPT	Probabilistic polynomial-time
PRF	Pseudorandom function
PRP	Pseudorandom permutation
RSA	Rivest Shamir Adleman
SHA	Secure Hash Algorithm
SQL	Structured Query Language
SSE	Searchable Symmetric Encryption
SSL	Secure Sockets Layer

Appendix B

Queries

In order to make the upcoming tables showing all search queries more compact, the Reuters1000 collection is denoted by R1, the Reuters2000 by R2 and so on. As in the main part of this thesis, the values in parentheses define the number of documents found for the single keywords being part of the respective multiple keyword query.

B.1. Single Keyword Queries

Search Query	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
government	91	156	226	309	405	504	596	693	792	889
loss	90	169	248	345	448	541	625	725	805	887
securities	56	122	213	291	371	442	523	610	686	758
meeting	53	101	152	199	262	310	354	422	483	569
industry	47	91	139	190	241	301	345	397	452	503
countries	33	79	114	153	191	232	270	338	383	441
payments	31	58	90	119	153	180	209	244	281	310
proposal	23	49	73	99	132	160	186	209	236	253
brazil	21	40	57	70	97	111	128	146	166	183
coffee	17	30	39	47	62	67	77	84	90	102
shipment	8	19	27	35	46	49	56	62	71	80
strategy	7	13	19	27	37	43	53	65	74	83
hardware	2	7	10	10	12	13	15	18	20	22
cancer	1	2	3	5	6	8	11	11	12	13
laboratory	0	1	5	5	8	9	9	11	13	14

Table B.1.: Document Matches for Single Keyword Queries.

B.2. Two Keyword Queries

Search Query	R1		R2		R3		R4	
loss, year	54	(90,269)	102	(169,562)	148	(248,798)	207	(345,1075)
government, year	44	(91,269)	79	(156,562)	100	(226,798)	134	(309,1075)
industry, year	27	(47,269)	43	(91,562)	70	(139,798)	94	(190,1075)
offering, securities	14	(43,56)	26	(83,122)	48	(139,213)	62	(180,291)
meeting, agreement	12	(53,85)	23	(101,161)	34	(152,239)	42	(199,311)
brazil, countries	8	(21,33)	19	(40,79)	20	(57,114)	23	(70,153)
crisis, payments	8	(13,31)	11	(23,58)	15	(32,90)	18	(40,119)
petroleum, energy	1	(18,12)	5	(36,31)	10	(49,56)	15	(61,86)
law, proposal	1	(11,23)	2	(27,49)	5	(47,73)	8	(66,99)
strategy, failure	1	(7,8)	2	(13,16)	2	(19,20)	3	(27,26)

(a) R1 - R4

Search Query	R5		R6		R7	
loss, year	261	(448,1376)	310	(541,1659)	358	(625,1925)
government, year	182	(405,1376)	231	(504,1659)	270	(596,1925)
industry, year	117	(241,1376)	145	(301,1659)	170	(345,1925)
offering, securities	70	(212,371)	85	(251,442)	103	(308,523)
meeting, agreement	54	(262,402)	59	(310,474)	65	(354,550)
brazil, countries	30	(97,191)	32	(111,232)	39	(128,270)
crisis, payments	23	(50,153)	25	(58,180)	29	(72,209)
petroleum, energy	22	(89,110)	24	(101,143)	27	(118,170)
law, proposal	8	(89,132)	8	(108,160)	8	(125,186)
strategy, failure	4	(37,39)	4	(43,42)	4	(53,47)

(b) R5 - R7

Search Query	R8		R9		R10	
loss, year	412	(725,2201)	463	(805,2478)	509	(887,2759)
government, year	308	(693,2201)	355	(792,2478)	392	(889,2759)
industry, year	191	(397,2201)	222	(452,2478)	251	(503,2759)
offering, securities	119	(361,610)	131	(412,686)	144	(453,758)
meeting, agreement	90	(422,659)	100	(483,747)	121	(569,846)
brazil, countries	46	(146,338)	53	(166,383)	59	(183,441)
crisis, payments	40	(87,244)	43	(96,281)	43	(109,310)
petroleum, energy	32	(141,194)	39	(167,231)	41	(195,252)
law, proposal	9	(146,209)	10	(162,236)	10	(170,253)
strategy, failure	4	(65,52)	5	(74,61)	5	(83,75)

(c) R8 - R10

Table B.2.: Document Matches for Two Keyword Queries.

B.3. Three Keyword Queries

Search Query	R1		R2		R3	
growth, government, year	12	(47,91,269)	20	(78,156,562)	23	(119,226,798)
investment, government,year	10	(63,91,269)	14	(98,156,562)	19	(153,226,798)
creditor, payments,debt	9	(16,31,74)	12	(25,58,139)	14	(31,90,230)
quotas, meeting,agreement	8	(16,53,85)	10	(28,101,161)	10	(35,152,239)
brazil, debt, president	7	(21,74,64)	8	(40,139,137)	13	(57,230,214)
growth, investment, government	6	(47,63,91)	10	(78,98,156)	13	(119,153,226)
volume, trading, year	5	(12,48,269)	7	(28,84,562)	8	(42,131,798)
outlook, economy, growth	4	(12,32,47)	7	(26,59,78)	9	(36,80,119)
dividend, assets, earnings	3	(32,28,35)	3	(63,64,70)	3	(92,96,100)
dollar, system, industry	2	(34,51,47)	3	(60,95,91)	3	(88,139,139)

(a) R1 - R3

Search Query	R4		R5		R6	
growth, government, year	29	(155,309,1075)	39	(207,405,1376)	45	(260,504,1659)
investment, government,year	20	(207,309,1075)	26	(262,405,1376)	34	(317,504,1659)
creditor, payments,debt	16	(35,119,299)	17	(44,153,390)	21	(52,180,454)
quotas, meeting,agreement	10	(41,199,311)	12	(51,262,402)	14	(59,310,474)
brazil, debt, president	13	(70,299,274)	19	(97,390,375)	20	(111,454,442)
growth, investment, government	14	(155,207,309)	20	(207,262,405)	23	(260,317,504)
volume, trading, year	9	(53,185,1075)	12	(73,241,1376)	13	(89,285,1659)
outlook, economy, growth	10	(45,101,155)	11	(53,130,207)	12	(61,155,260)
dividend, assets, earnings	3	(134,117,150)	3	(165,158,201)	3	(204,197,247)
dollar, system, industry	3	(120,173,190)	3	(155,225,241)	3	(184,279,301)

(b) R4 - R6

Table B.3.: Document Matches for Three Keyword Queries.

Search Query	R7		R8	
growth, government, year	55	(303,596,1925)	63	(356,693,2201)
investment, government,year	41	(366,596,1925)	48	(431,693,2201)
creditor, payments,debt	24	(58,209,538)	28	(70,244,624)
quotas, meeting,agreement	14	(62,354,550)	15	(65,422,659)
brazil, debt, president	22	(128,538,532)	27	(146,624,617)
growth, investment, government	25	(303,366,596)	32	(356,431,693)
volume, trading, year	14	(96,332,1925)	16	(110,388,2201)
outlook, economy, growth	13	(66,178,303)	16	(80,219,356)
dividend, assets, earnings	4	(247,237,293)	4	(275,266,335)
dollar, system, industry	3	(211,332,345)	3	(251,398,397)

(c) R7 - R8

Search Query	R9		R10	
growth, government, year	70	(403,792,2478)	75	(446,889,2759)
investment, government,year	57	(487,792,2478)	64	(547,889,2759)
creditor, payments,debt	31	(76,281,708)	33	(81,310,774)
quotas, meeting,agreement	16	(69,483,747)	21	(78,569,846)
brazil, debt, president	35	(166,708,704)	36	(183,774,783)
growth, investment, government	34	(403,487,792)	37	(446,547,889)
volume, trading, year	19	(123,446,2478)	22	(142,511,2759)
outlook, economy, growth	20	(90,246,403)	21	(94,280,446)
dividend, assets, earnings	5	(311,315,387)	5	(346,368,429)
dollar, system, industry	3	(299,445,452)	3	(375,449,503)

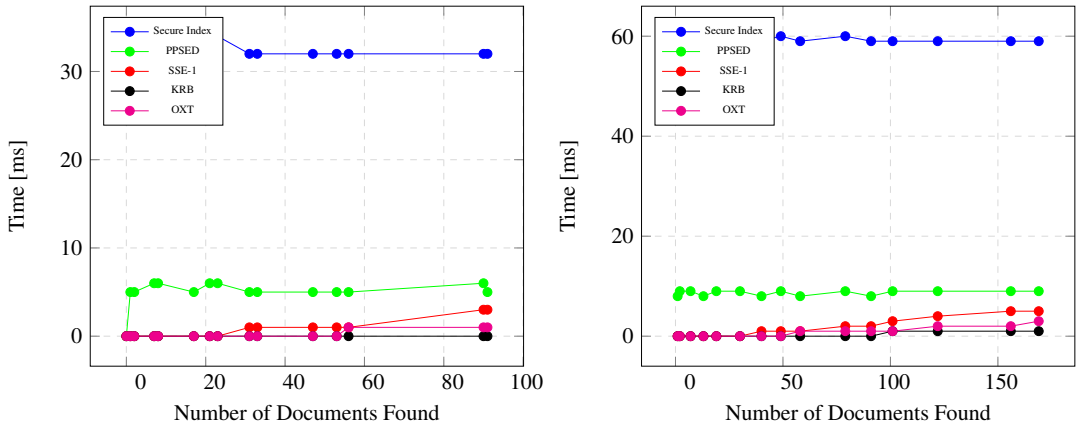
(d) R9 - R10

Table B.3.: Document Matches for Three Keyword Queries (continued).

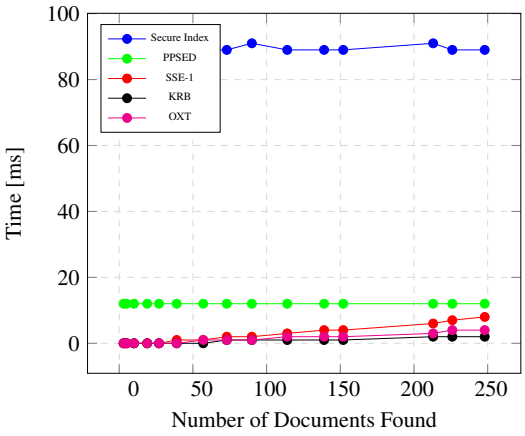
Appendix C

Experimental Results

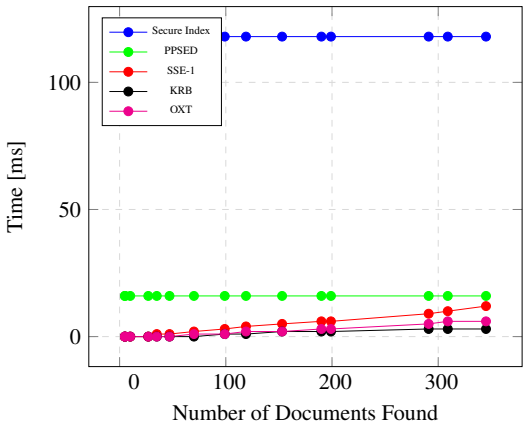
C.1. Single Keyword Searches



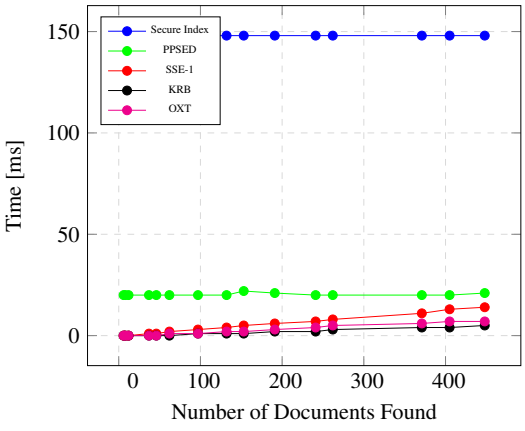
(a) Reuters1000 **(b)** Reuters2000



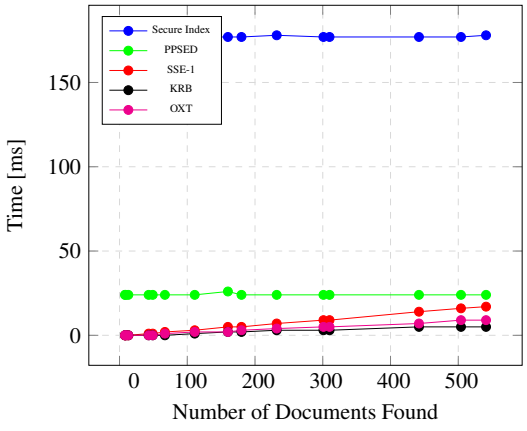
(c) Reuters3000



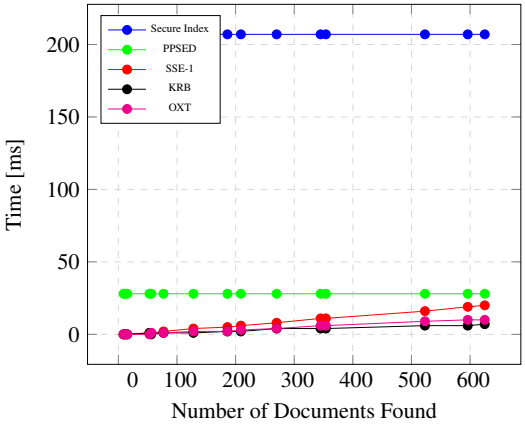
(d) Reuters4000



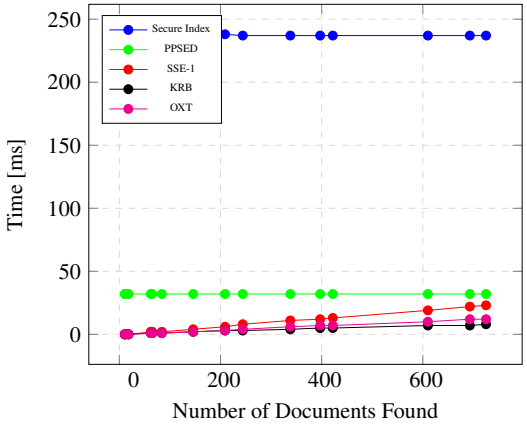
(e) Reuters5000



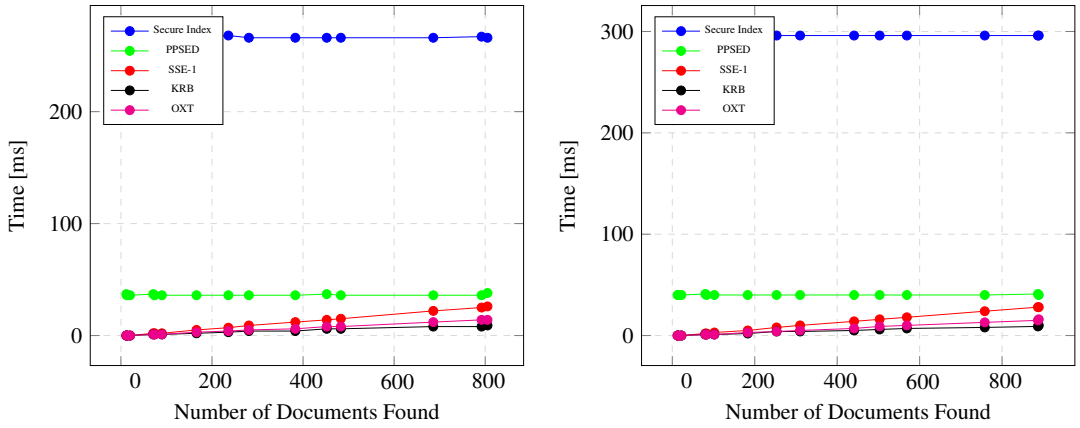
(f) Reuters6000



(g) Reuters7000



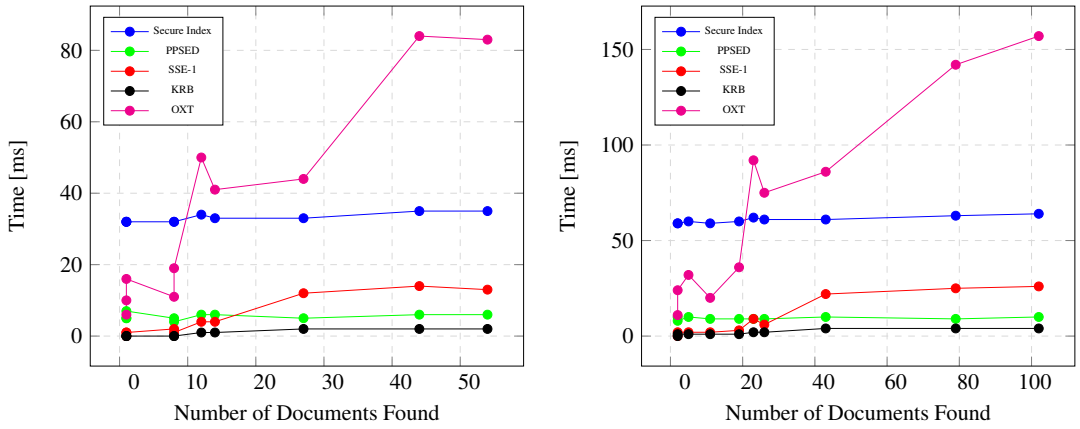
(h) Reuters8000



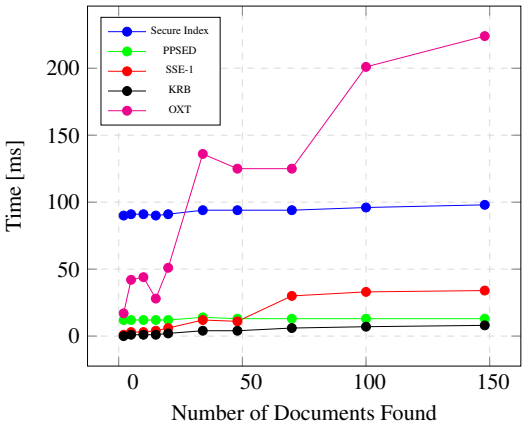
(i) Reuters9000 **(j)** Reuters10000

Figure C.-1.: Single Keyword Searches on All Document Collections.

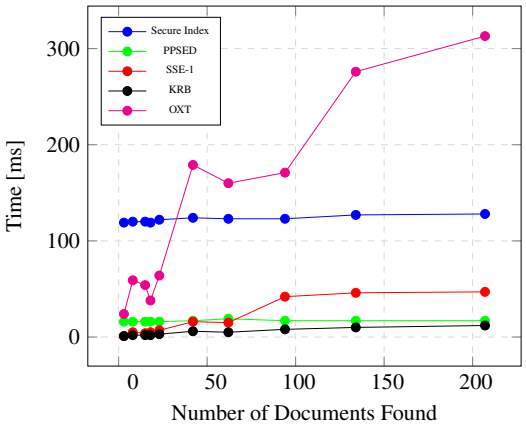
C.2. Two Keyword Searches



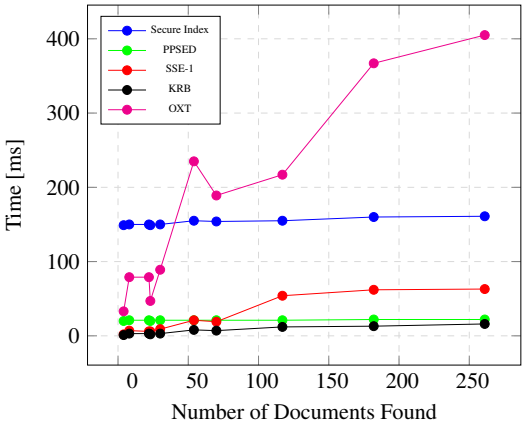
(a) Reuters1000 **(b)** Reuters2000



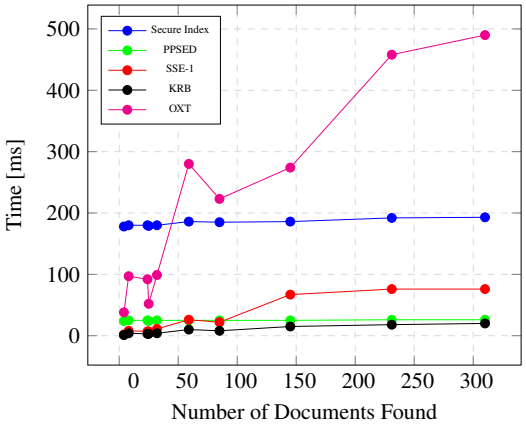
(c) Reuters3000



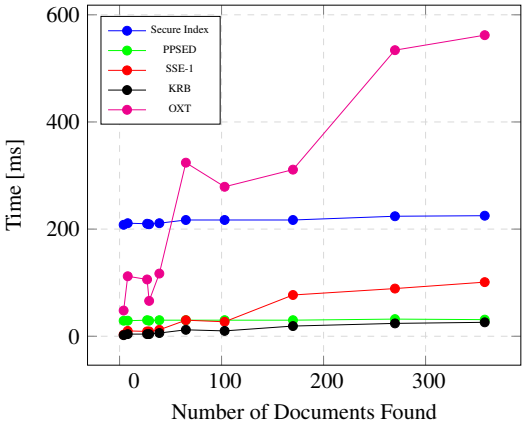
(d) Reuters4000



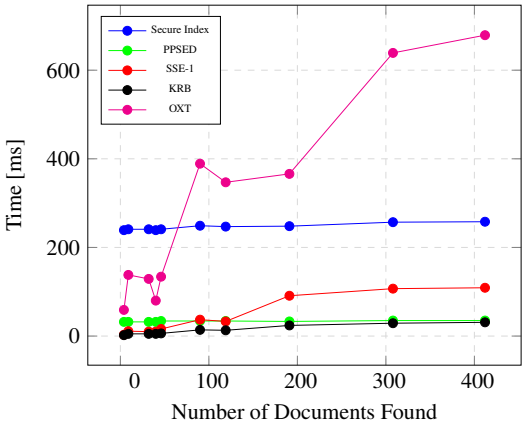
(e) Reuters5000



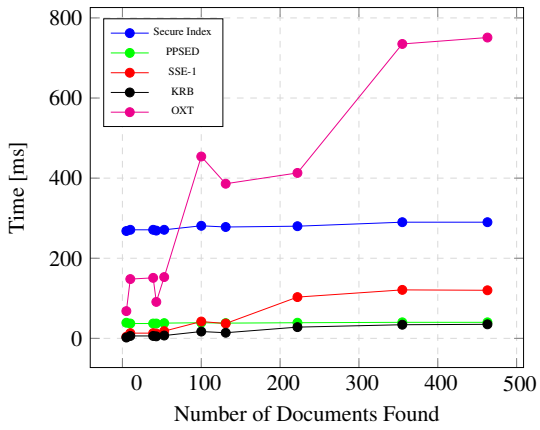
(f) Reuters6000



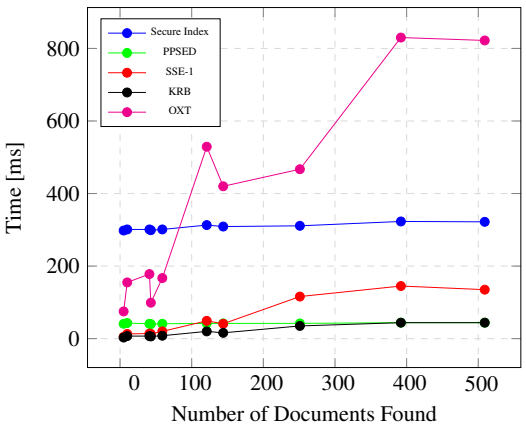
(g) Reuters7000



(h) Reuters8000



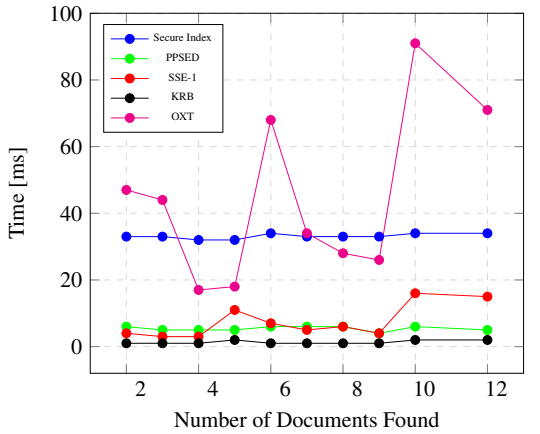
(i) Reuters9000



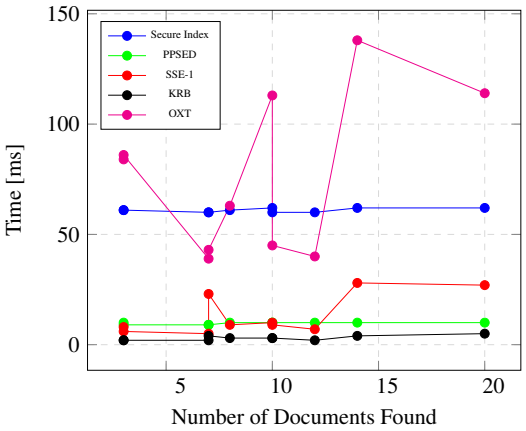
(j) Reuters10000

Figure C.-2.: Two Keyword Searches on All Document Collections.

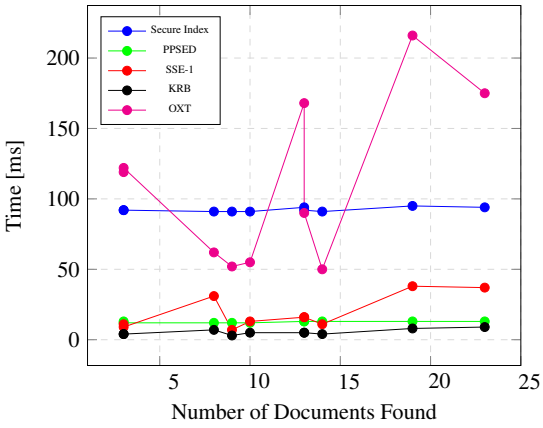
C.3. Three Keyword Searches



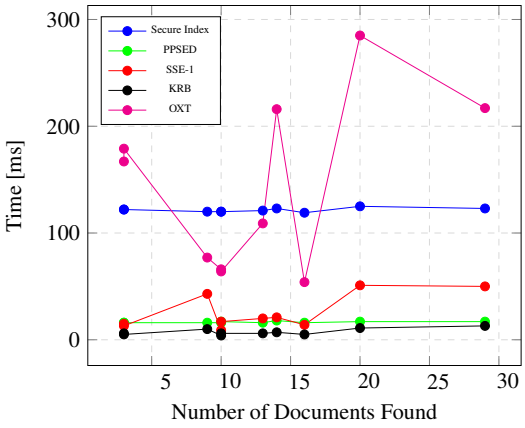
(a) Reuters1000



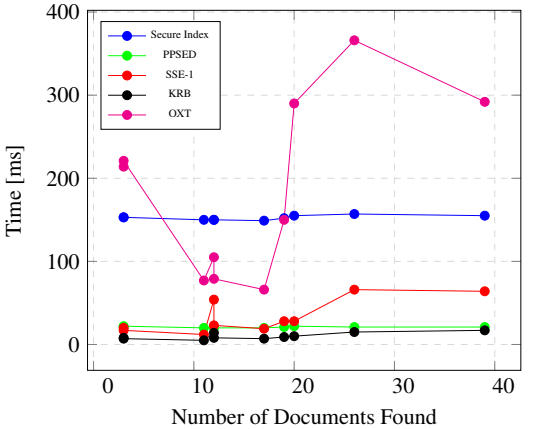
(b) Reuters2000



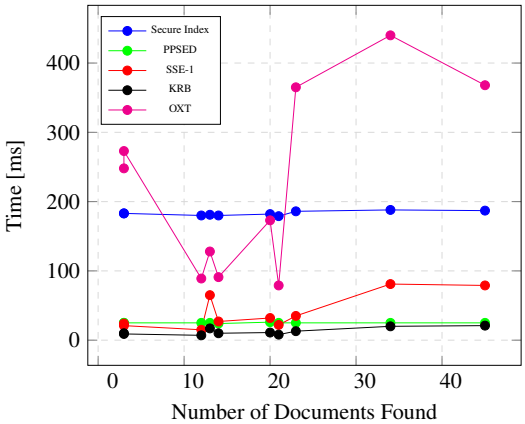
(c) Reuters3000



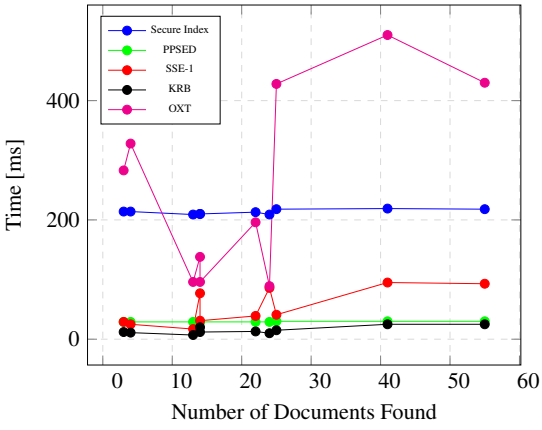
(d) Reuters4000



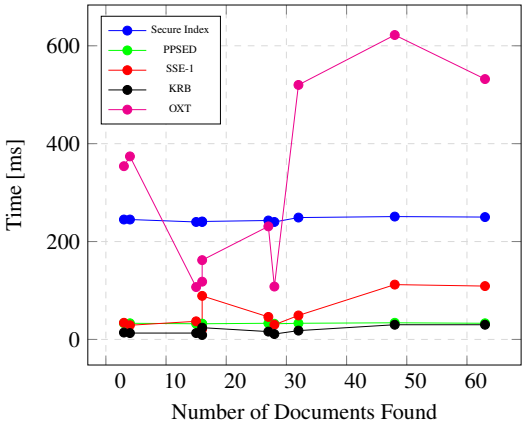
(e) Reuters5000



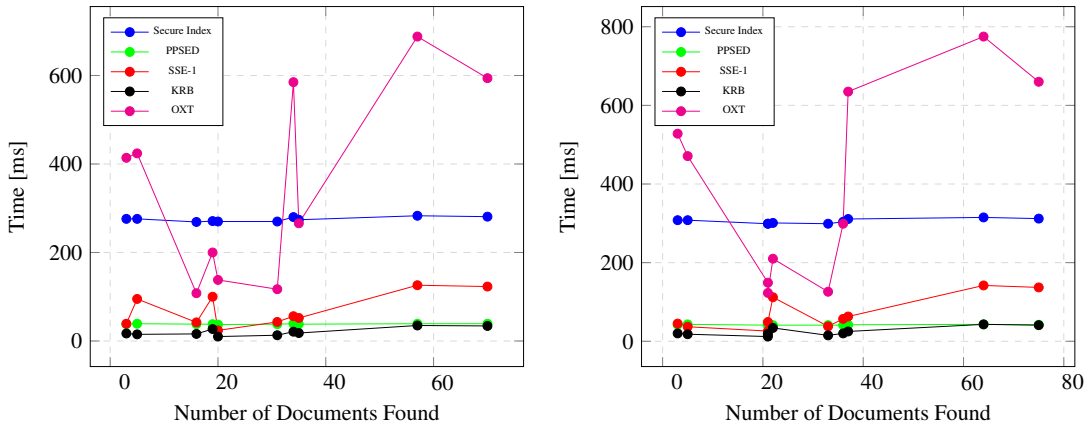
(f) Reuters6000



(g) Reuters7000



(h) Reuters8000



(i) Reuters9000 **(j)** Reuters10000

Figure C.-3.: Three Keyword Searches on All Document Collections.